PALS: Efficient Or-Parallel Execution of Prolog on Beowulf Clusters

K. Villaverde and E. Pontelli
Dept. Computer Science
New Mexico State University
Las Cruces, NM
{kvillave,epontell}@cs.nmsu.edu

H. Guo
Dept. Computer Science
University of Nebraska at Omaha
Omaha, NE
haifengguo@mail.unomaha.edu

G. Gupta
Dept. Computer Science
University of Texas at Dallas
Richardson, TX
gupta@utdallas.edu

October 17, 2003

Abstract

This paper describes the development of the *PALS* system, an implementation of Prolog capable of efficiently exploiting or-parallelism on *distributed memory* platforms—specifically Beowulf clusters. PALS makes use of a novel technique, called *incremental stack-splitting.* The technique proposed builds on the stack-splitting approach, previously described by the authors and experimentally validated on shared-memory systems in [18, 38], which in turn is an evolution of the stack-copying method used in a variety of parallel logic and constraint systems—e.g., MUSE, YAP, and Penny [1, 30, 25]. This is the first distributed implementation based on the stack-splitting method ever realized, and the results presented confirm the superiority of this approach as a simple but yet effective technique to transition from shared-memory to distributed memory systems. PALS extends stack-splitting by combining it with incremental copying; the paper provides a description of the implementation of PALS, including details of how distributed scheduling is handled. We also investigate methodologies to effectively support order-sensitive predicates (e.g., side-effects) in the context of the stack-splitting scheme. Experimental results obtained from running PALS on both Shared Memory and Beowulf system are presented and analyzed.

1 Introduction

The literature on parallel logic programming (e.g., see [19]) underscores the potential for achieving excellent speed-ups and performance improvements from execution of logic programs on parallel architectures, with little or no programmer intervention. Particular attention has been devoted over the years to the design of technology for supporting *or-parallel* execution of Prolog programs on shared-memory architectures.

Or-parallelism (OP) arises from the non-determinism implicit in the process of reducing a given subgoal using different clauses of the program. The non-deterministic structure of a logic programming execution is commonly depicted in the form of a *search tree* (a.k.a. *or-tree*). Each internal node represents a *choice-point*, i.e., an execution point where multiple clauses are available to reduce the selected subgoal. Leaves of the tree represent either failure points (i.e., resolvents where the selected subgoal does not have a matching clause) or success points (i.e., solutions to the initial goal). A sequential computation boils down to traversal of this search tree according to some predefined search strategy. Languages like Prolog mostly adopt a fixed strategy (a left-to-right, depth-first traversal strategy), while others may adapt their search strategy according to the structure of each computation.

While in a sequential execution the multiple clauses that match a subgoal are explored one at a time via backtracking, in or-parallel execution they are explored concurrently using multiple execution threads (*computing agents*). These multiple agents traverse the or-tree looking for unexplored branches. If an unexplored branch (i.e., an untried clause to resolve a selected subgoal) is found, the agent picks it up and begins execution. This agent will stop either if it fails (reaches a failing leaf), or if it finds a solution. In case of failure, or if the solution found is not acceptable to the user, the agent will *backtrack*, i.e., move back up in the tree, looking for other choice-points with untried alternatives to explore. The agents may need to synchronize if they access the same node in the tree—to avoid repetition of computations.

Intuitively, or-parallelism allows concurrent search for solution(s) to the original goal. The importance of the research on efficient techniques for handling or-parallelism arises from the generality of the problem—technology originally developed for parallel execution of Prolog programs has found application in contexts such as constraint programming (e.g., [32, 26]) and non-monotonic reasoning (e.g., [11, 27]). Efficient implementation of or-parallelism has also been extensively investigated in the context of AI systems [22, 23].

In sequential implementations of search-based AI systems or Prolog, typically one branch of the tree resides on the inference engine's stacks at any given time. This simplifies implementation quite significantly. However, in case of parallel systems, multiple branches of the tree co-exist at the same time, making parallel implementation quite complex. Efficient management of these co-existing branches is quite a difficult problem, and it is referred to as the *environment management problem* [15].

Most research in or-parallel execution of Prolog so far has focused on techniques aimed at *shared-memory multiprocessors (SMPs)*. Relatively fewer efforts [12, 3, 8, 7, 6, 33] have been devoted to implementing Prolog systems on *distributed memory platforms (DMPs)*. Out of these efforts only a small number have been implemented as working prototypes, and even fewer have produced acceptable speedups. Various implementations rely on out-of-date and fairly specialized architectures, such as transputers [3, 6, 7], while others have been developed only as simulators [33]. Existing techniques developed for SMPs are inadequate for the needs of DMP platforms. In fact, most implementation methods require sharing of data and control stacks to work correctly. Even if the need to share data stacks is eliminated—as in *stack copying* [1] or in *stack recomputation* [9]—the need to share parts of the control stack still exists. The control stack can be large, thus degrading the performance on DMPs.

Experimental [1] and theoretical studies [28] have demonstrated that *stack-copying*, and in particular *incremental* stack-copying, is one of the most effective implementation techniques devised for exploiting or-parallelism. Stack-copying allows sharing of work between parallel agents by copying the state of one agent (which owns unexploited tasks) to another agent (which is currently idle). The idea of *incremental* stack-copying is to only copy the *difference* between the state of two agents, instead of copying the entire state each time. Incremental stack-copying has been used to implement or-parallel Prolog efficiently in a variety of systems (e.g., MUSE [1], YAP [30]), as well as to exploit parallelism from constraint systems [31] and non-monotonic reasoning systems [27, 11].

In order to further reduce the communication during stack-copying and make its implementation efficient on DMPs, we propose a new technique, called *stack-splitting* [18, 38]. In this paper, we describe stack-splitting in detail, and provide results from the first ever concrete implementation of stack-splitting on both shared-memory multiprocessors (SMPs) and distributed-memory multiprocessors (DMPs)—specifically, a Pentium-based Beowulf—along with a novel scheme to combine incremental copying with stack-splitting on DMPs. The *incremental stack-splitting* scheme is based on a procedure which labels parallel choice-points and then compares the labels to determine the fragments of data and control areas that need to be exchanged between agents. We also describe scheduling schemes suitable for our incremental stack-splitting scheme and variations of stack-splitting providing efficient handling of order-sensitive predicates (e.g., side-effects). Both the incremental stack-splitting and the scheduling schemes described have been implemented in the *PALS* system, a message-passing or-parallel implementation of Prolog. In this paper we present performance results obtained from this implementation. To our knowledge, PALS is the first ever or-parallel implementation of Prolog realized on a Beowulf architecture (built from off-the-shelf components). The techniques we propose are immediately applicable to other systems based on the same underlying model, e.g., constraint programming [31] and non-monotonic reasoning [27] systems.

In the rest of this work we will focus on execution of Prolog programs (unless explicitly stated otherwise); this means that we will assume that programs are executed according to the computation and selection rules of Prolog. We will also frequently use the term *observable semantics* to indicate the overall observable behavior of an execution—i.e., the order in which all visible activities of a program execution take place (order of input/output, order in which solutions are obtained, etc.). If a parallel computation respects the observable Prolog semantics, then this means that the user does not see any difference between such computation and a sequential Prolog execution of the same program. Clearly, our goal is to develop parallel execution models that properly reproduce Prolog's observable semantics.

The rest of the paper is organized as follows. Section 2 provides an overview of the main issues related to or-parallel execution of Prolog. Section 3 describes the stack-splitting scheme, while Section 4 describes its implementation. Section 5 analyzes the problem of guaranteeing efficient distribution of work between idle processors. Section 6 offers a general discussion about possible optimization to the implementation of stack-splitting. Section 7 describes how stack-splitting can be adapted to provide efficient handling of order-sensitive predicates of Prolog (e.g., control constructs, side-effects). Section 8 analyzes the result obtained from the prototype implementation in the PALS system. Finally, Section 9 provides conclusions and directions for future research.

2 Or-Parallelism

2.1 Foundations of Or-Parallelism

Parallelization of logic programs can be seen as a direct consequence of Kowalski's principle [21]

$$\textit{Programs} = \textit{Logic} + \textit{Control}$$

Program development separates the control component from the logical specification of the problem, thus making the two orthogonal. The lack (or, at least, the limited presence) of knowledge about control in the program allows the run-time systems to adopt different execution strategies without affecting the declarative meaning of the program (i.e., the set of logical consequences of the program). The same is true of search-based systems, where the order of exploration of the branches of the search-tree is flexible (within the limits imposed by the semantics of the search strategy—e.g., search heuristics).

Apart from the separation between logic and control, from a programming languages perspective, logic programming offers two key features which make exploitation of parallelism more practical than in traditional imperative languages:

1. From an operational perspective, logic programming languages are *single-assignment* languages; variables are mathematical entities which can be assigned a value at most once during each derivation—this relieves a parallel system from having to keep track of complex flow dependencies such as those needed in parallelization of traditional programming languages [42].

2. The operational semantics of logic programming, unlike imperative languages, makes substantial use of *non-determinism*—which in turn can be easily converted into parallelism without radically modifying the overall operational semantics. Furthermore, control in most logic programming

languages is largely implicit, thus limiting programmers' influence on the development of the flow of execution.

The second point is of particular importance: the ability to convert existing non-determinism into parallelism leads to the possibility of extracting parallelism directly from the execution model without any modification to the language (*implicit parallelization*).

Or-parallelism arises when more than one rule defines a relation and a subgoal unifies with more than one rule head—the corresponding bodies can then be executed in parallel with each other, giving rise to or-parallelism. Or-parallelism is thus a way of efficiently searching for solutions to the query, by exploring in parallel the search space generated by the presence of multiple clauses applicable at each resolution step. Observe that each parallel computation is potentially computing a distinct solution to the original goal.

Or-parallelism frequently arises in applications that explore a large search space via backtracking. This is the typical case in application areas such as expert systems, constraint optimization problems, parsing, natural language processing, and scheduling. Or-parallelism also arises during parallel execution of deductive database systems [13, 41].

2.2 The Environment Representation Problem

A major problem in implementing OP is that multiple branches of the search tree are active simultaneously, each of which may produce a solution or fail. Each of these branches may bind a variable created earlier—i.e., before the choice-point—with a different value. In normal sequential execution, where only one branch of the search tree is active at any time, the binding for the variable created by that branch can be stored directly in the memory location allocated for such variable. During backtracking, this binding is removed, so as to free the memory location for use by the next branch.

However, in OP execution, this memory location will have to be turned into a *set of locations*, shared between processors, or some other means devised to store the multiple bindings that may *simultaneously* exist. In addition, we should be able to efficiently distinguish bindings applicable to each branch during variable access. The problem of maintaining multiple bindings efficiently is called the *multiple environments representation* problem [28, 19].

We can illustrate this with a simple example. Consider a goal of the type `?- p(X)` and let us assume that the program contains two clauses for `p`:

```
p(a) :- ....
p(b) :- ....
```

During an OP execution the two clauses can be explored by different processors. Then the variable `X` receives a different binding from each of the two clauses. These bindings need to be represented and maintained separately, as they will each lead to a different solution (`X=a` and `X=b`) for the initial goal. `X=a` and `X=b` represent different instances of `X` and they need to be separately maintained and properly associated to the correct thread of execution—i.e., the correct branch in the or-tree.

One of the main issues, thus, in designing an implementation scheme capable of efficiently supporting or-parallelism is the development of an efficient way of associating the correct set of bindings to each branch of the or-tree. The naive approach of keeping a complete separate copy of the answer substitution for each branch is highly inefficient, since it requires the creation of complete copies of the substitution (which can be arbitrarily large) every time a branching takes place [19, 28]. A large number of different methodologies have been proposed to address the environment representation problem in OP [19].

Variations of the same problem arise in many classes of search problems and non-deterministic programming. For example, in the context of non-monotonic reasoning under stable models semantics [14, 27], the computation needs to determine the possible belief sets; these are determined by guessing the truth values of selected predicates and deriving the consequences of such guess. In this case, the dynamic environment is represented by the truth values of the various predicates along each branch of the tree. The environment representation problem also arises in parallel heuristic search.

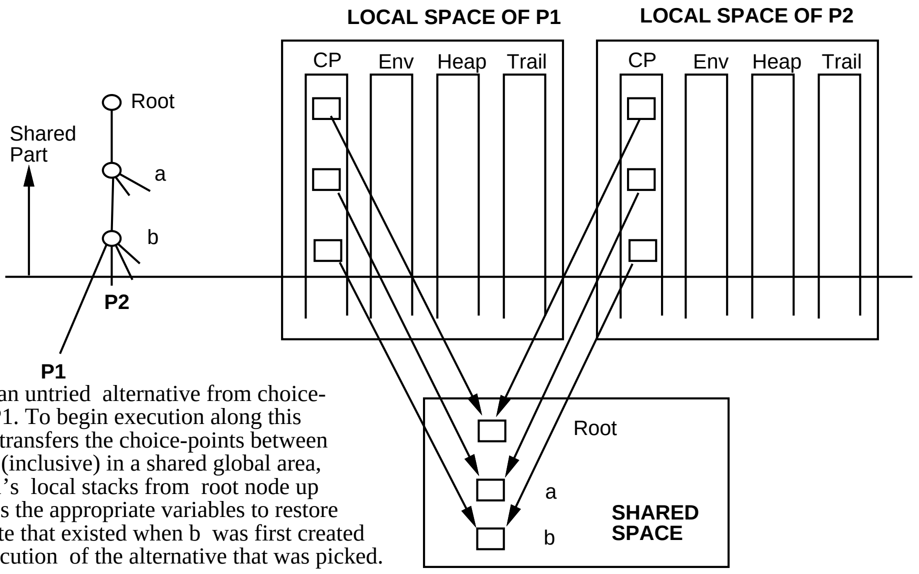
2.3 Stack-copying for Maintaining Multiple Environments

Stack-copying [1] is a successful approach for environment representation in OP. In this approach (originally developed in the MUSE system), processors maintain a *separate* but *identical* address space. Whenever a processor $\mathcal{A}$ becomes idle (*idle-agent*), it will start looking for unexplored alternatives generated by another processor $\mathcal{B}$ (*active-agent*). Once a choice-point p is detected in the tree $\mathcal{T}_{\mathcal{B}}$ generated by $\mathcal{B}$, $\mathcal{A}$ will create a local copy of $\mathcal{T}_{\mathcal{B}}$ and restart the computation by backtracking over p. Since all or-agents maintain an identical logical address space, the creation of a local copy of $\mathcal{T}_{\mathcal{B}}$ is reduced to a simple memory copying (Fig. 1)—without the need for any explicit pointer relocation. Since each or-agent owns a separate copy of the environments, the environment representation problem is readily solved—each or-agent will store the locally produced bindings in the local copy of the environments. Additionally, each or-agent performs Prolog execution on a private copy of its tree branch, thus relieving the need for sharing memory. For this reason, stack-copying has been considered highly suitable for execution on DMPs, where stack-copying can be simply replaced with message passing between processors.

In practice, the stack-copying operation is more involved than simple memory copying, as the choice-points have to be copied to an area accessible to all processors. This is important because the set of untried alternatives is now shared between the two processors; if this set is not accessed in mutual exclusion, then two processors may execute the same alternative. Thus, after copying, each choice-point in $\mathcal{T}_{\mathcal{B}}$ will be transferred to a shared area—these will be called *shared frames*. Both active and idle agents will replace their choice-points with pointers to the corresponding shared frames. Shared frames are accessed in mutual exclusion. This whole operation of obtaining work from another processor is usually termed *sharing of or-parallel work*. This is illustrated in Figure 1; the part of the tree labeled as *shared* has been copied from processors `P1` to processor `P2`; the choice-points lying in this part of the tree have been also moved to the shared space to avoid repetition of work.

In order to reduce the number of sharing operations, unexplored alternatives are always picked from the bottom-most choice-point in the tree [1, 5]. During the sharing operation all the choice-points between the bottom-most and the top-most choice-points are shared between the two processors. This means that, in each sharing operation, we try to maximize the amount of work shared between the two processors. Furthermore, to reduce the amount of information transferred during the sharing operation, copying is done *incrementally*, i.e., only the *difference* between $\mathcal{T}_{\mathcal{A}}$ and $\mathcal{T}_{\mathcal{B}}$ is actually copied.

A major reason for the success of MUSE [5] and YAP [30] is that they effectively implement incremental stack copying with *scheduling on bottom-most choice-point*. Each idle processor picks work from the bottom-most choice-point of an or-branch. The stack segments upwards of this choice-point are copied before the exploration of this alternative is begun. The copied stack segments may contain other choice-points with untried alternatives—which become accessible via simple backtracking. Thus, a significant amount of work becomes available to the copying processor every time a sharing operation is performed. The cost of having to copy potentially larger fragments of the tree becomes relatively insignificant considering that this technique drastically reduces the *number of sharing operations* per-



Processor P2 picks an untried alternative from choice-point b created by P1. To begin execution along this alternative, P2 first transfers the choice-points between the root node and b (inclusive) in a shared global area, and then copies P1's local stacks from root node up to node b. It untrails the appropriate variables to restore the computation state that existed when b was first created and begins the execution of the alternative that was picked.

Figure 1: Stack-copying based Or-parallelism

formed. It is important to observe that each sharing operation requires both the processors involved to stop the regular computation and cooperate in the sharing.

3 The Split Choice-point Stack-Copying Model

3.1 Copying on DMPs

During the copying operation, the choice-points—or at least part of their content—are transferred to a shared memory area, to make sure that the same alternative is not tried by two processors maintaining copies of the same stack-segment. By moving the choice-points to the shared area, this problem is avoided, as a processor has to acquire exclusive access to the choice-point in the shared memory area before it can pick an untried alternative from it.

Whenever backtracking encounters a shared choice-point, most information needed has to be retrieved by acquiring exclusive access to the corresponding shared frame. This solution works fine on SMPs—where mutual exclusion is easily implemented using *locks*. However, on a DMP this process is a source of significant overhead—access to the shared area becomes a bottleneck [4]. This is because sharing of information in a DMP leads to frequent exchange of messages and hence considerable overhead. Centralized data structures, such as the shared frames, are expensive to realize in a distributed setting. Nevertheless, stack copying has been considered by most authors as the best environment representation methodology to support OP in a distributed memory setting [8, 12, 3, 7, 6]. This is because, while the choice-points are shared, at least other data-structures representing the computation—such as, in the case of Prolog, the environment, the trail, and the heap—are not. Other environment representation schemes, e.g., the popular Binding Arrays scheme [24], have been specifically designed for SMPs and share most of the computation; the communication overhead produced by these alternative schemes on DMPs is likely to be prohibitive.[1] To avoid the problem of sharing choice-points in distributed implementations, many implementors have reverted back to the *scheduling on top-most choice-point*

[1]Some recent works have proposed to combine these methods with distributed shared memory schemes [33].

strategy [8, 12]. The reason is that untried alternatives of a choice-point created higher up in the or-tree are more likely to generate large subtrees, and sharing work from the highest choice-point reduces the amount of duplicated work as well as leads to smaller-sized stacks being copied. However, if the granularity does not turn out to be large, then another untried alternative has to be picked and a new copying operation has to be performed. In contrast, in scheduling on bottom-most, more work could be found via backtracking, since more choice-points are copied during the same sharing operation. Additionally, scheduling on bottom-most is closer to the depth-first search strategy used by sequential systems, and facilitates support of Prolog semantics.

Research done on comparing scheduling strategies indicates that scheduling on bottom-most is superior to scheduling on top-most [5]. This is especially true for the stack-copying technique because:

1. the number of copying operations is minimized; and,
2. the alternatives in the choice-points copied are "cheap" sources of additional work, available via backtracking.

However, the fact that these choice-points are shared is a major drawback for a distributed implementation of copying. The question we consider is: can we avoid sharing of choice-points while keeping scheduling on bottom-most? The answer is affirmative, as is discussed next.

3.2 Split Choice-point Stack Copying

In the Stack-Copying, the primary reason why a choice-point has to be shared is because we want to serialize the selection of untried alternatives, so that no two processors can pick the same alternative. The shared frame is locked while the alternative is selected to achieve this effect. However, there are other simple ways of ensuring the same property: *the untried alternatives of a choice-point can be split between the two copies of the choice-point stack.* We call this operation *choice-point stack-splitting* or simply *stack-splitting.* This will ensure that no two processors pick the same alternative.

We can envision different schemes for splitting the set of alternatives between shared choice-points—e.g., each choice-point receives half of the alternatives, or the partitioning can be guided by information regarding the unexplored computation, such as granularity and likelihood of failure. In addition, the need for a shared frame, as a critical section to protect the alternatives from multiple executions, has disappeared, as each stack copy has a choice-point with a different set of unexplored alternatives. All the choice-points can be evenly split in this way during the copying operation.

The choice-point stack-splitting operation is illustrated in figure 2. The strategy adopted in this example is what we call *horizontal splitting*: the remaining alternatives in each of the shared choice-point are split between the two agents.

Another simple strategy that one can adopt is *vertical splitting.* Each processor is given all the alternatives of alternate choice-points (See Figure 3). Thus, in this case, the alternatives are not split, rather the list of choice-points available is split between the two processors. In fact, this idea of splitting the list of choice-points may even be more efficient than splitting alternatives. Splitting of alternatives can be resorted to when very few choice-points are present in the stack. Vertical splitting is particularly useful when the search tree is *binary*—which is a frequent situation in both Prolog as well as other search problems (e.g., non-monotonic reasoning where the choice-points represent choices of truth values).

Different mixes of splitting of the list of choice-points and choice-point splitting can be tried to achieve a good load balance—as discussed in [36, 34]. Eventually, the user can also be given control regarding how the splitting is done.

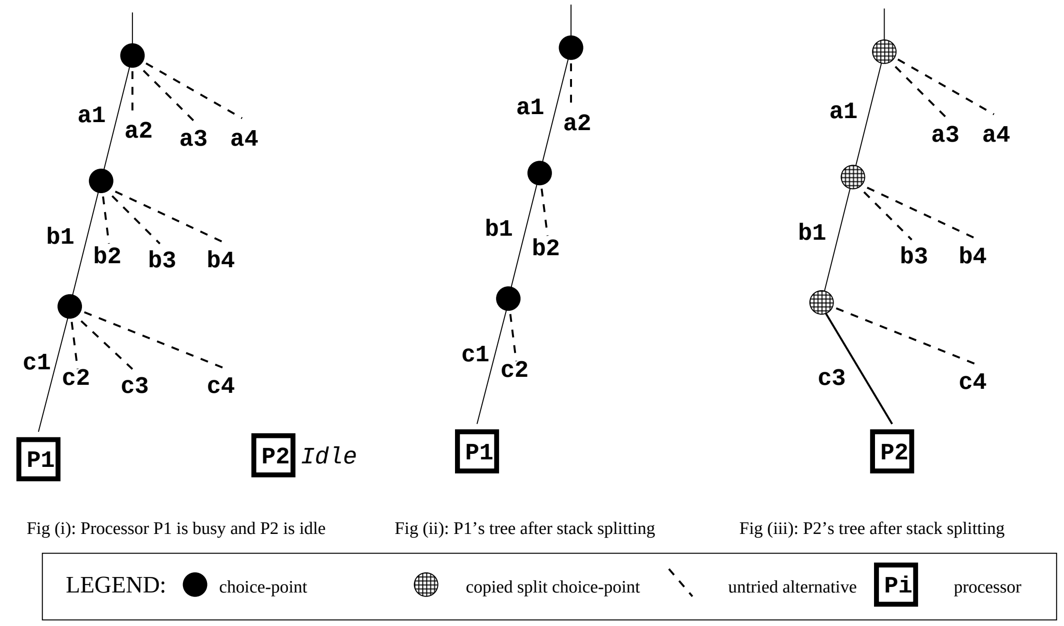


Fig (i): Processor P1 is busy and P2 is idle Fig (ii): P1's tree after stack splitting Fig (iii): P2's tree after stack splitting

Figure 2: Stack-splitting based or-parallelism

The major advantage of stack-splitting is that scheduling on bottom-most can still be used without incurring huge communication overheads. Essentially, after splitting the different or-parallel threads become independent of each other, and hence communication is minimized during execution. This makes the stack-splitting technique highly suitable for DMPs. The possibility of parameterizing the splitting of the alternatives based on additional semantic information (granularity, non-failure, user annotations) can further reduce the likelihood of additional communications due to scheduling.

3.3 Incremental Stack-Splitting

Traditional stack-copying requires agents which share work to transfer a complete copy of the data structures representing the status of the computation. In the case of a Prolog computation, this may include transferring most of the choice-points along with copies of the other data areas (trail, heap, environments). Since Prolog computations can make use of large quantities of memory (e.g., generate large structures on the Heap), this copying operation can become quite expensive. Existing stack-copying systems (e.g., MUSE) have introduced a variation of stack-copying, called *Incremental Stack-Copying* [1] which allows to considerably reduce the amount of data transferred during a sharing operation. The idea is to compare the content of the data areas in the two agents involved in a sharing operation, and transfer only the difference between the state of the two agents. This is illustrated in Fig. 4. In Fig. 4(i) we have two agents (P1 and P2) which have 3 choice-points in common (e.g., from a previous sharing operation). P1 owns two additional choice-points with unexplored alternatives while P2 is out of work. If P2 obtains work from P1, then there is no need of copying again the 3 top choice-points (Fig. 4(ii)).

Incremental stack-copying, in a shared-memory context, is relatively simple to realize—the shared frames can be used to identify which choice-points are in common and which are not [1].

In the rest of the paper we describe a complete implementation of stack-splitting using a message passing platform, analyzing in detail how the various problems mentioned earlier have been tackled. In addition to the basic stack-splitting scheme, we analyze how stack-splitting can be extended to incorporate *incremental copying*, an optimization which has been deemed essential to achieve speed-

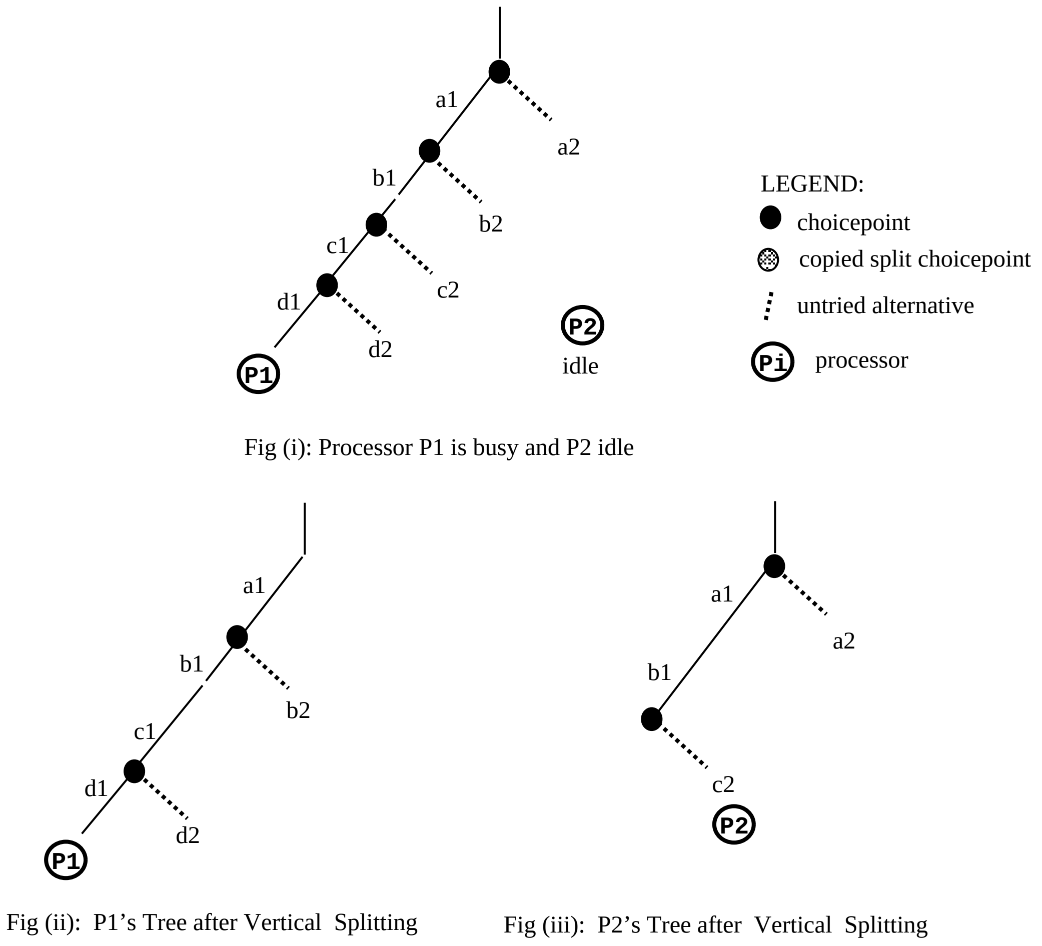

a1
a2
b1
b2
c1
c2
d1
d2
P1
P2
idle
LEGEND:
choicepoint
copied split choicepoint
untried alternative
Pi
processor
Fig (i): Processor P1 is busy and P2 idle
Fig (ii): P1's Tree after Vertical Splitting
Fig (iii): P2's Tree after Vertical Splitting

Figure 3: Vertical Splitting of Choice-Points

ups in various classes of benchmarks, and to handle order-sensitive predicates (e.g., side-effects). The solution we describe has been developed in a concrete implementation, realized by modifying the engine of a commercial Prolog system (ALS Prolog) and making use of the Message Passing Interface (MPI) as communication platform. The ALS Prolog system is based on an efficient version of the Warren Abstract Machine (WAM).

4 Implementation Description

4.1 Data Structures

During stack-splitting, all WAM data areas (the WAM model includes a stack for the choice-points, a stack for environments, a heap for the dynamic creation of terms, and a trail used to support undoing of variable binding during backtracking), except for the code area, are copied from the agent giving work to the idle one. Next, the parallel choice-points are split between the two agents. Blindly copying all the stacks every time an agent shares work with another idle agent can be wasteful, since frequently the two agents already have parts of the stacks in common due to previous copying. We can take advantage of this fact to reduce the amount of copying by performing *incremental copying*, as discussed earlier. In incremental copying, only the difference between the stacks of the agent that is giving work and the idle

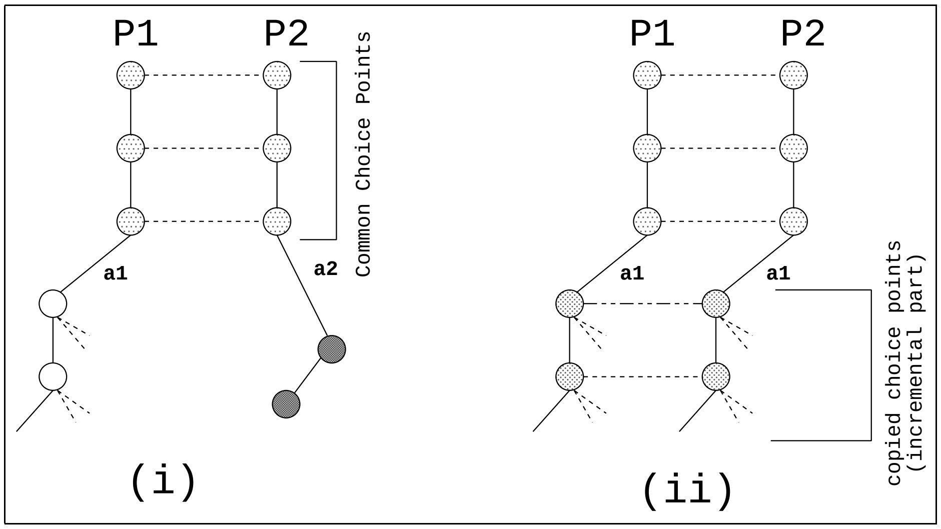


Figure 4: Incremental Stack-Copying

agent is copied on to the memory areas of the idle agent. Incremental copying has been implemented in the MUSE and YAP systems with relatively little overhead. This is primarily because all the information needed for performing incremental copying efficiently can be found in the shared frames—the use of shared frames is essential to determine the bottom-most *common* choice-point between the two agents. The determination of such choice-point is typically accomplished by analyzing the bitmaps stored in the various shared frames, which are used to keep track of agents which own a copy of the associated choice-point. In our stack-splitting scheme, there are no shared frames, hence performing incremental stack-copying will incur more overhead due to the communication overhead involved.

In order to figure out the incremental part that only needs to be copied during incremental stack-splitting, parallel choice-points will be *labeled* in a certain way. The goal of the labeling process is to uniquely identify the original "source" of each choice-point (i.e., which agent created it), to allow unambiguous detection of copies of common choice-points. Thus, the labels effectively replace the bitmaps used in the shared memory implementations of stack-copying. The labeling procedure is described next.

To perform labeling, each agent maintains a counter. Initially, the counter in each agent is set to *1*. The counter is increased by 1 every time the labeling procedure is performed. When a parallel choice-point is copied for the first time, a label for it is created. The label is composed of three parts:

1. agent rank,
2. counter, and
3. choice-point address.

The agent rank is the rank (i.e., id) of the agent which created the choice-point. The counter is the current value of the labeling counter for the agent generating the labels. The choice-point address is the address of the choice-point which is being labeled. The labels for the parallel choice-points are recorded in a separate *label stack*, in the order they are created. Also, when a parallel choice-point is removed from the stack, its corresponding label is also removed from the label stack. Initially, the label stack in each agent is set to *empty*. Intuitively, the label stack keeps a record of changes done to the stacks since the last stack-splitting operation. Observe that the choice of maintaining labels in a stack—instead of associating them directly to the corresponding choice-points—has been dictated by efficiency reasons.

Let us illustrate the stack-splitting accompanied by labeling with an example. Suppose process A has just created two parallel choice-points and process B is idle. Process A and B have their counters set to *1* and their label stacks set to *empty*. Then Process B requests work from process A. Process A first creates labels for its two parallel choice-points. These labels have their rank and counter parts as *A:1*. Process A then pushes these labels into its label stack. See Figure 5 (for simplicity in our figures, we do not show the label stack explicitly but show each label rank and counter parts inside the parallel choice-point being labeled). Notice that process A incremented its counter to 2 after the labeling procedure was over.

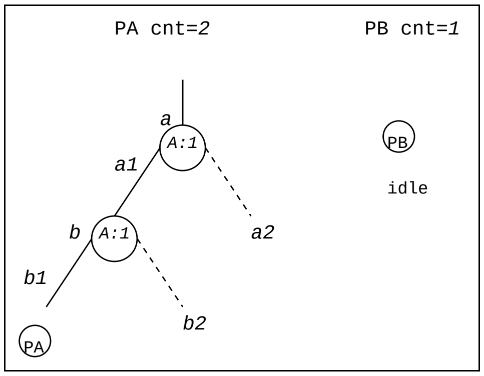


Figure 5: Process A Labels its two Parallel Choice-points

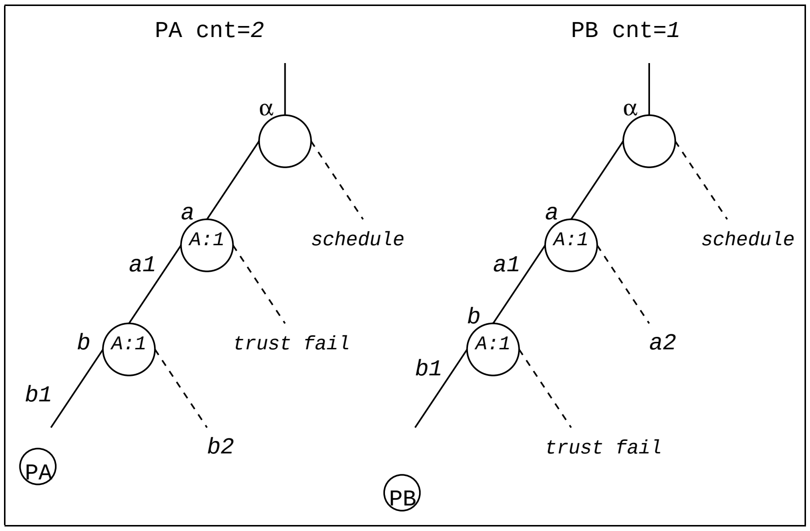


Figure 6: Process A Gave Work to Process B

Stack-copying is done next. Process B gets all the parallel choice-points of process A along with process A label stack. Then, stack-splitting takes place: process A will keep the alternative *b2* but not *a2*, and process B will keep the alternative *a2* but not *b2*. We have designed a new WAM scheduling instruction which is placed in the next alternative field of the choice-point above which there is no more parallel work. This scheduling instruction implements the scheduling scheme described in Section 5. Process A keeps the alternative *b2* of choice-point *b*, changes the next alternative field of choice-point *a* to WAM instruction *trust_fail* to avoid taking the original alternative of this choice-point, and changes the next alternative field of the choice-point above *a* to the new WAM instruction *schedule* which will take process A into scheduling. Process B changes the next alternative field of choice-point *b* to WAM instruction *trust_fail* to avoid taking the original alternative of this choice-point, keeps the alternative *a2* of choice-point *a*, and changes the next alternative field of the choice-point above *a* to the *schedule* instruction. See Figure 6. Afterwards, process B backtracks, removes choice-point *b* along with its corresponding label in the label stack, and then takes alternative *a2* of choice-point *a*.

Suppose now that process B creates two parallel choice-points and process C is idle. Process C, with its counter set to *1* and its label stack set to *empty*, requests work from B. Process B first creates labels for its two new parallel choice-points. These labels have their rank and counter parts as *B:1*. Process B then pushes these labels into its label stack. See Figure 7. Notice that process B incremented its counter to *2*.

Now, stack-copying takes place. Process C gets all the parallel choice-points of process B along with process B label stack. Stack-splitting takes place: process B will keep the alternative *d2* but not *c2*,

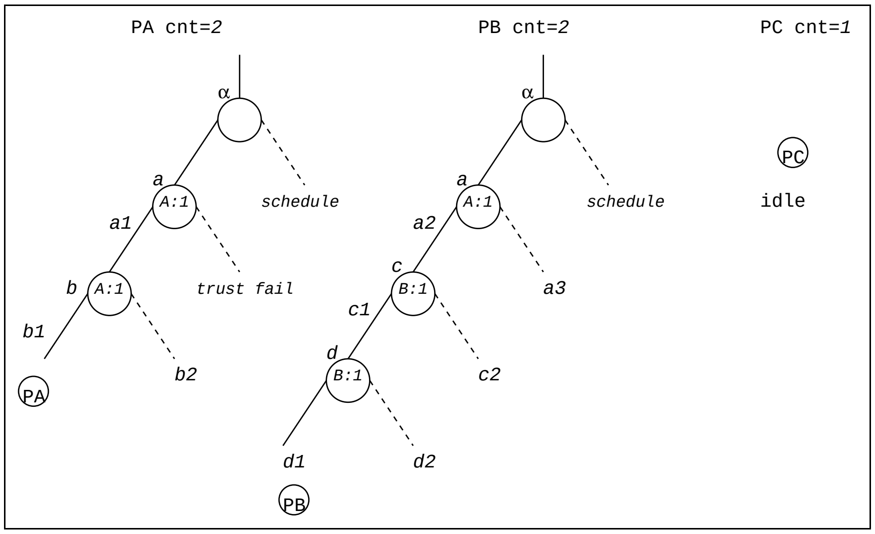


Figure 7: Process B Labels its Two New Parallel Choice-points

and process C will keep the alternative *c2* but not *d2*. We have modified stack-splitting so that this activity takes place only in the recently created, labeled, and never split parallel choice-points of the process giving work – e.g., *c* and *d*. In other words, the process receiving work should only get new work from these parallel choice-points. At this point in time it is possible to perform stack-splitting: process B will keep alternatives *d2* and *a3* but not *c2*, and process C will keep alternative *c2* but not *d2* nor *a3*. Notice that all three parallel choice-points of process B have been split among B and C. Process B keeps the alternative *d2* of choice-point *d* and changes the next alternative field of choice-point *c* to WAM instruction *trust_fail* to avoid taking the original alternative of this choice-point, and keeps the alternative *a3* of choice-point *a*. Process C changes the next alternative field of choice-point *d* to WAM instruction *trust_fail* to avoid taking the original alternative of this choice-point, keeps the alternative *c2* of choice-point *c*, changes the next alternative field of choice-point *a* to WAM instruction *trust_fail* (to avoid taking the original alternative of this choice-point), and changes the next alternative field of the choice-point above *a* to *schedule*. This is illustrated in Figure 8. Process C backtracks, removes choice-point *d* along with its corresponding label in the label stack, and then takes alternative *c2* of choice-point *c*.

4.2 Incremental Stack-splitting: The Procedure

In this section we describe how the label stacks are used to compute the incremental part to be copied. Assume process W is giving work to process I. Process W will label all its parallel choice-points which have not been labeled before and will push them into its label stack. Process W then increments its counter.

If process I label stack is empty, then stack-copying will need to be performed followed by stack-splitting. Process W sends its complete choice-point stack and its complete label stack to process I. Then stack-splitting is performed on all the parallel choice-points of process W. Process I then tries its new work via backtracking.

However, if process I label stack is not empty then process I sends its label stack to process W. Process W compares its label stack against the label stack from I. Comparison will go from the first label entered in the stacks to the last label entered in the stacks. The objective is to find the last choice-point *ch* with a common label. In this way, processes W and I are guaranteed to have the same computation

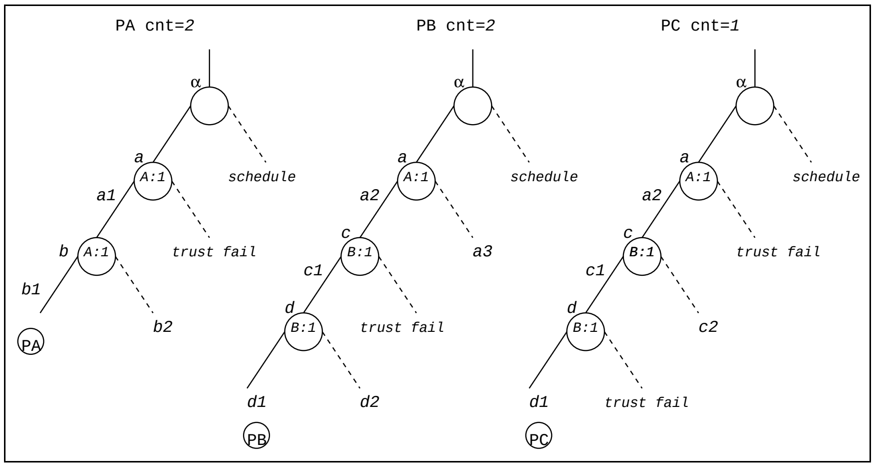


Figure 8: Process B Gives Work to Process C

above the choice-point *ch*, while their computations will be different below such choice-point.

If the choice-point *ch* does not exist, then (non-incremental) stack-copying will need to be performed followed by stack-splitting just as described before. However, if choice-point *ch* does exist, then process I backtracks to choice-point *ch*, and performs incremental-copying. Process W sends its choice-point stack starting from choice-point *ch* to the top of its choice-point stack. Process W also sends its label stack starting from the label corresponding to choice-point *ch* to the top of its label stack. Stack-splitting is then performed on all the parallel choice-points of process W. Afterwards, agent I tries its new work via backtracking.

We illustrate the above procedure by the following example. Suppose process A has three parallel choice-points and process C requests work from A. Process A first labels its last two parallel choice-points which have not been labeled before and then increments its counter. Afterwards, process C sends its label stack to process A. Process A compares its label stack against the label stack of process C and finds the last choice-point *ch* with a common label. Above choice-point *ch*, the Prolog trees of processes A and C are equal. Below choice-point *ch*, the Prolog trees of processes A and C differ. See Figure 9.

Now, process C backtracks to choice-point *ch*. Incremental stack-copying can then take place. Process A sends its choice-point stack starting from choice-point *ch* to the top of its choice-point stack. Process A also sends its label stack starting from the label corresponding to choice-point *ch* to the top of its label stack. Then, stack-splitting takes place on the three parallel choice-points of process A. See Figure 10. Process C backtracks to choice-point *i* and takes alternative *i2*.

4.3 Incremental Stack-splitting: Challenges

Four issues that were not discussed above and which are fundamental for the correct implementation of the incremental stack-splitting scheme presented are discussed below.

4.3.1 Sequential Choice-points

The first issue is related to the management of *sequential choice-points*. Typically, only a subset of the choice-points present during the execution are suitable to provide work that can be effectively parallelized. These choice-points are traditionally called *parallel choice-points*, to distinguish them from *sequential choice-points*, whose alternatives are meant to be explored by a single processor.

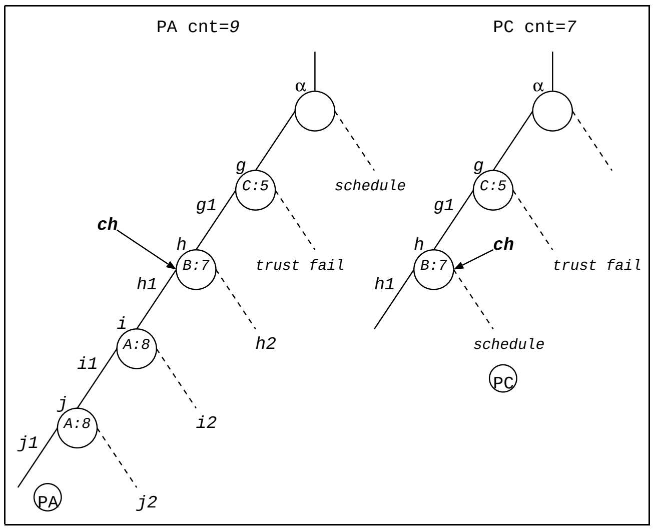


Figure 9: Process A Labels its Two New Parallel Choice-points and Compares Labels with Process C

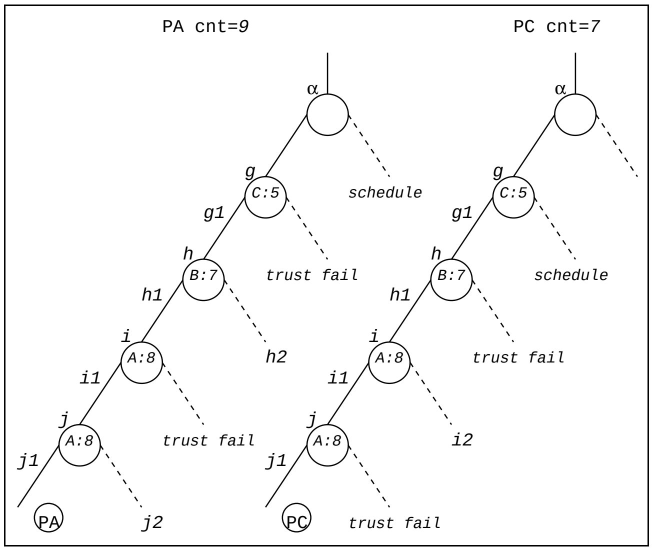


Figure 10: Process A Gave Work to Process C

The problem arises when sequential choice-points are located among the parallel choice-points that will be split between two agents. If the alternatives of these choice-points are kept in both processes, we may have repeated, useless or wrong computations. Hence, the alternatives of these choice-points should only be kept in one process (e.g., the process that is giving work), or these choice-points should also be split. If the alternatives are kept in the process giving work, then the process that is receiving work should change the next alternative field of all these choice-points to the WAM instruction *trust_fail* to avoid taking the original alternatives of these choice-points.

4.3.2 Installation Process

The second issue has to do with the bindings of conditional variables (i.e., variables that may be bound differently in different or-parallel branches) which need to be copied too as part of the incremental stack-splitting process. For example, suppose that in our last example before process A gives work to process C, process A created variable X before choice-point *ch* was created and instantiated it after choice-point *ch* was created. See Figure 11. We can see that the binding for X was not copied during incremental stack-splitting. This is because X is a conditional variable which was created before choice-point *ch* and the incremental part of the heap or environment stack that was copied did not contain its binding.

Therefore, we need to modify our procedure so that it also performs the installation of the bindings of all conditional variables. This can be done by having the process giving work create when necessary a stack of all these conditional variables along with their bindings. This stack will then be sent to the process receiving work so that it can perform the bindings of all these conditional variables.

4.3.3 Garbage Collection

The third issue arises when garbage collection takes place. When this happens, relocation of choice-points may also take place. Hence, the labels in our label stack may no longer label the correct parallel

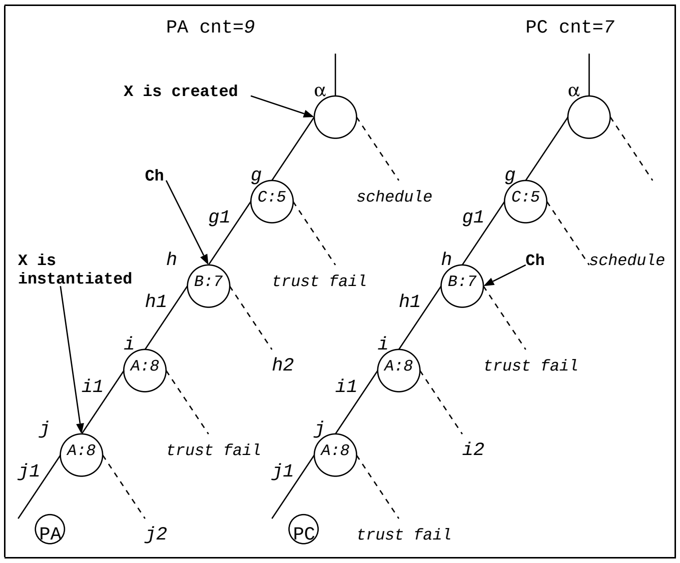


Figure 11: The Binding of Conditional Variable X Needs to be Copied

choice-points. Therefore, we need to modify our labeling procedure so that when garbage collection on an agent takes place, the label stack of this agent is invalidated. This can be done by just setting its label stack to empty. The next time this process gives work, full stack-copying will have to take place. This solution is analogous to the one adopted in the MUSE system [1].

4.3.4 Next Clause Fields

The fourth issue arises when the next clause fields of the parallel choice-points between the first parallel choice-point *first cp* and the last choice-point *ch* with a common label in the agent giving work are not the same compared to the ones in the agent receiving work. This situation occurs after several copying and splitting operations. In this case, we can not just copy the part of the choice-point stack between choice-point *ch* and the top of the stack and then perform the splitting. This is because the splitting will not be performed correctly. For example, suppose that in our previous example when process C requests work from process A, we have this situation. See Figure 12.

We can see that choice-point g should be given to process C. But process C does not have the right next clause field for this choice-point. Hence, we need to modify our procedure once again. This can be done by having the process giving work send all the next clause fields between its first parallel choice-point *first cp* and choice-point *ch* to the process receiving work. Then the splitting of all parallel choice-points can take place correctly. See Figure 13.

5 Scheduling

Scheduling is an important aspect of any parallel system. The scheduling strategy adopted largely determines the level of speed-up obtained for a particular parallel execution. The main objective of a scheduling strategy is to balance the amount of parallel work done by different agents. Additionally, work distribution among agents should be done with a minimum of communication overhead. These two goals are somewhat at odds with each other, since achieving perfect balance may result in a very complex scheduling strategy with considerable communication overhead, while a simple scheduling strategy which re-distributes work less often will incur low communication overhead but poor balancing of work.

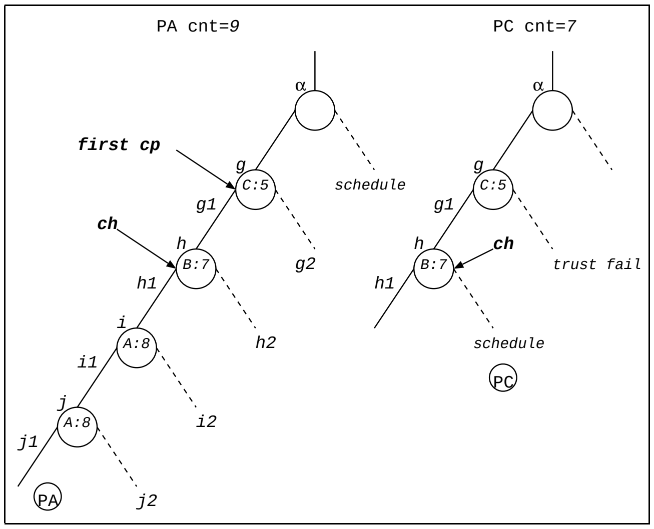


Figure 12: Copy the Next-clause Fields between *first cp* and *ch*

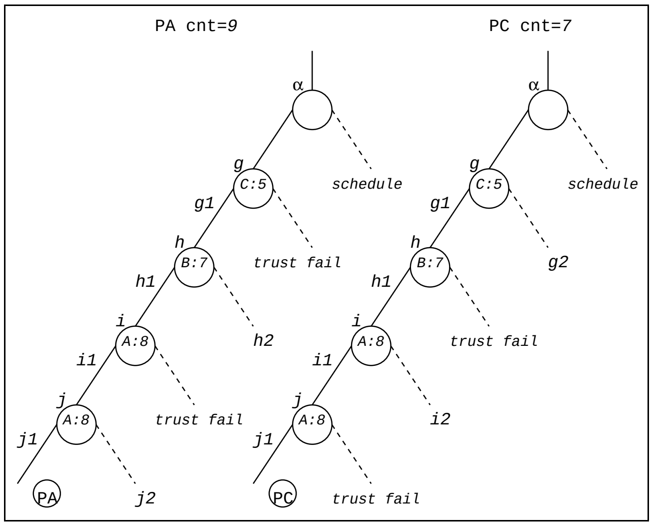


Figure 13: Process C Received the Next Clause Fields

Therefore, it is obvious that there is an intrinsic contradiction between distributing parallel work as even as possible and minimizing the distribution overhead. Thus our main goal is to find a trade-off point that results in a reasonable scheduling strategy.

We adopt a simple and fair distributed algorithm to implement a scheduling strategy in the PALS system. A new data structure—the *load vector*—is introduced to indicate the work loads of different agents. The work load of an agent is approximated by the number of parallel choice-points present in its local computation tree. Each agent keeps a work load vector *V* in its local memory, and the value of *V[i]* represents the work load of the agent with rank *i*. Based on the work load vector, an idle agent can request parallel work from other agent with the greatest work load, so that parallel work can be fairly distributed. The load vector is updated at runtime. When stack-splitting is performed, a `Load_Info` message with updated load information will be broadcasted to all the agents so that each agent has the latest information of work load distribution. Additionally, load information is attached with each incoming message. For example: when a `Request_Work` message is received from agent P_1, the value of P_1's work load, 0, can be inferred.

Based on its work load each agent can be in one of two states: *scheduling* state or *running* state. When an agent has some work to do, it is in a running state, otherwise, it is in a scheduling state. An agent that is running, occasionally checks whether there are incoming messages. Two possible types of messages are checked by the running agent: one is `Request_Work` message sent by an idle agent, and the other is `Send_Load_Info` message, which is sent when stack-splitting occurs. The idle agent in scheduling state is also called a scheduling agent. An idle agent wants to get work as soon as possible from another agent, preferably the one that has the largest amount of work. The scheduling agent searches through its local load vector for the agent with the greatest work load, and then sends a `Request_Work` message to that agent asking for work. If all the other agents have no work, then the execution of the current query is finished and the agent halts. When a running agent receives a `Request_Work` message, stack-splitting will be performed if the running agent's work load is greater than the splitting threshold, otherwise, a `Reply_Without_Work` message with a positive work load value will be sent as a reply. If a scheduling agent receives a `Request_Work` message, a `Reply_Without_Work` message with work load 0 will be sent as a reply.

The distributed scheduling algorithm mainly consists of two parts: one is for the scheduling agent, and the other is for the running agent. The running agent's algorithm can be briefly described as follows:

```
while (any incoming message) {
   status = the message's status;
   switch (status.TAG) {
   case Send_LoadInfo:
        update the corresponding agents' work load.
        break;
   case Request_Work:
        if (local work load > Splitting Threshold) {
           stack-splitting with the status.SOURCE agent;
           broadcast the updated work load to all the agents;
        }
        else
           reply a message with the tag Reply_Without_Work
                 and the value of its own work load;
        break;
   }
}
```

At fixed time intervals (which can be selected at initialization of the system) the agent examines the content of its message queue for eventual pending messages. `Send_LoadInfo` messages are quickly processed to update the local view of the overall load in the system. Messages of the type `Request_Work` are handled as described above. Observe that the concrete implementation actually checks for the presence of the two types of messages with different frequency (i.e., request for work messages are considered less frequently than requests for load update).

The scheduling agent's algorithm can be briefly described as follows:

```
while (1) {
   D = the rank of the agent with the greatest work load;
   if (D's work load == 0) exit;   /* The whole work is done */
   Send a Request_Work message to D;
   matched = 0;
   While (!matched) {
      status = the message's status;
      switch (status.TAG) {
         case Reply_With_Work:
              stack-splitting with the status.SOURCE agent;
              update the corresponding work load;
              simulate failure and go to execute the split work;
              break;
         case Reply_Without_Work:
              if (status.SOURCE == D) matched = 1;
              V[status.SOURCE] = work load of status.SOURCE;
              break;
         case Request_Work:
              reply a message with the tag Reply_Without_Work and
                   its work load 0 to status.SOURCE;
              V[status.SOURCE] = 0;
              break;
         case Send_LoadInfo:
              update the corresponding agents' work load;
```

```
                    break;
                }
            }
```

6 Costs and Optimizations

6.1 Overheads of Stack-Splitting

The shared frame used in the regular stack-copying technique is also a place where global information related to scheduling is kept. The shared frames provide a globally accessible description of the or-tree, and each shared frame keeps information regarding which processor is working in which part of the tree. This last piece of information is needed to support the kind of scheduling typically used in stack-copying systems—work is taken from the processor that is "closer" in the computation tree, thus reducing the amount of information to be copied—since the difference between the stacks is minimized. The shared nature of the frames ensures accessibility of this information to all processors, providing a consistent picture of the computation.

However, under stack-splitting the shared frames no longer exist; scheduling and work-load information will have to be maintained in some other way. They could be kept in a global shared area similar to the case of SMPs—e.g., by building a representation of the or-tree—or distributed over multiple processors and accessed by message passing in case of DMPs. The maintenance of global scheduling information represents a problem which is orthogonal to the environment representation. This means that scheduling management in a DMP will anyway require communication between processors.

Shared frames are also employed in MUSE [1] to detect the Prolog order of choice-points, needed to execute order-sensitive predicates (e.g., side-effects, extra-logical predicates) in the correct order. As in the case of scheduling, some information regarding global ordering of choice-points needs to be maintained to execute order-sensitive predicates in the correct order—see Section 7. Thus, stack-splitting does not completely remove the need of a shared description of the or-tree. The use of stack-splitting can mitigate the impact of accessing shared resources—e.g., stack-splitting allows scheduling on bottom-most which, in general, leads to a reduction of the number of calls to the scheduler.

6.2 The Cost of Stack-Splitting

The stack-copying operation in Stack-Splitting is slightly more involved than in regular stack-copying. In MUSE, the original choice-point stack is traversed and the choice-points transferred to the shared area. This operation involves only those choice-points that have never been shared before—shared choice-points already reside in the global shared area. For this reason the actual *sharing* of the choice-points is performed by the *active-agent*—which is forced to interrupt its regular computation to assist the sharing process. The actual copying of the stack takes place only after the choice-points have been copied to the shared memory-area.

In the stack-splitting technique, the sharing is replaced first by a phase of splitting, performed by the active agent; after the copying is done, the idle agent will also traverse the copied choice-points, completing the splitting of the untried alternatives. In the case of SMP implementations, this operation is expected to be considerably cheaper than transferring the choice-points to the shared area. The actual splitting can be represented by a simple pair of indices that refer to the list of alternatives—which, in a SMP system like MUSE, is static and shared by all the processors. In the case of DMP implementations, the situation is similar: since each processor maintains a local copy of the code, the splitting can be performed by communicating to the copying processor which alternatives it can execute for each choice-point (e.g., a pair of pointers to the list of alternatives). It is simple to encode such information within the choice-point itself during copying.

In both cases we expect the sharing operation to have comparable complexity; a slight delay may occur in stack-splitting, due to the traversal of the choice-point stack performed by each processor. On the other hand, in stack-splitting the two traversals—one in the *idle-agent* and one in the *active-agent*—can be overlapped, while in the MUSE scheme the *idle-agent* is suspended until the *active-agent* has completed the sharing operation. However, if the stack being copied, S_o, is itself a copy of some other stack, then unlike regular stack-copying, we may still need to traverse both the source and target stacks and split the choice-points. The presence of this additional step depends on the policy adopted for the partitioning of the alternatives between processors. It is, for example, required if we adopt a policy which assigns half of the alternatives to each of the processors. In such cases, the cost of sharing will be slightly more than the cost of regular stack-copying.

Once a processor selects new work, it will look for work again only after it finishes the exploration of all alternatives acquired via stack-splitting. Incremental copying and other optimizations developed for MUSE may still apply to stack-splitting. E.g., processors should not immediately deallocate shared areas on backtracking, to allow for incremental copying.

6.3 Optimizing Stack-splitting Cost

The cost incurred in splitting the untried alternatives between the copied stack and the stack from which the copy is made, can be eliminated by amortizing it over the operation of picking untried alternatives. Let us assume that the untried alternatives will be split evenly (Fig. 16(ii)).

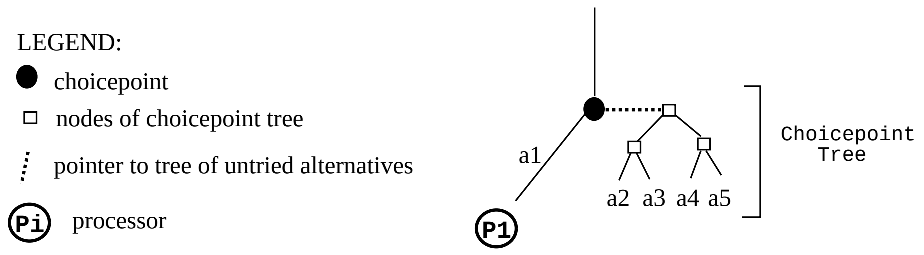

LEGEND:
choicepoint
nodes of choicepoint tree
pointer to tree of untried alternatives
Pi
processor
a1
a2
a3
a4
a5
P1
Choicepoint Tree

Figure 14: Amortizing Splitting Overhead

In the modified approach, no traversal and modification of the choice-points is done during copying. The untried alternatives are organized as a binary tree (see Figure 14). The binary alternatives can be efficiently maintained in an array, using standard techniques found in any data-structure textbook. In addition, each choice-point maintains the "copying distance" from the very first original choice-point as a bit string. This number is initially 0 when the computation begins. When stack-splitting takes place and a choice-point whose bit string is n is copied from, then the new choice-point's bit string is $n1$ (1 tagged to bit string n), while the old choice-point's bit string is changed to $n0$ (0 tagged to bit string n). When a processor backtracks to a choice-point, it will use its bit string to navigate in the tree of untried alternatives, and find the alternatives that it is responsible for. For example, if the bit-string of a processor is 10, then all the alternatives in the left subtree of the right subtree of the or-tree are to be executed by that processor.

However, it is not very clear which of the two strategies—incurring cost of splitting at copying time *vs* amortizing the cost over the selection of untried alternatives—would be more efficient. In case of amortization, the cost of picking an alternative from a choice-point is now slightly higher, as the binary tree of choice-points needs to be traversed to find the right alternative.

6.4 Applicability and Effectiveness of Stack-Splitting

Stack-splitting essentially performs static work distribution, as the untried alternatives are split at the time of picking work. If the choice-points that are split are balanced, then we can expect good performance. Thus, we should expect to see good performance when the choice-points generated by the computation that are parallelized contain a large number of alternatives. This is the case for applications which fetch data from databases and for most generate & test type of applications.

For choice-points with a small number of alternatives, stack-splitting is more susceptible to problems created by the static work distribution strategy that implicitly results from it: for example, in cases where OP is extracted from choice-points with only two alternatives. Such choice-points arise quite frequently, from the use of predicates like member and select:

```
member(X,[X|_]).                         select(X,[X|Y],Y).
member(X,[_|Y]) :- member(X,Y).          select(X,[Y|Z],[Y|R]) :- select(X,Z,R).
```

Both these predicates generates choice-points with only two alternatives—thus, at the time of sharing, a single alternative is available in each choice-point. The different alternatives are spread across different choice-points. Stack-splitting would assign all the alternatives to the copying processor, thus leaving the original processor without local work. However, the problems raised by such situations can be solved using a number of techniques:

- Use knowledge about the inputs and partial evaluation, or automatic optimizations (e.g., *Last Alternative Optimization (LAO)* [16]) to collapse the different choice-points into a single one.
- Use more complex splitting strategies, e.g., if a choice-point has odd number of untried alternatives remaining $(2n + 1)$, then one processor will be assigned n alternatives and the other $n + 1$. The processor which gets n and the processor which gets $n + 1$ can be alternated for the different choice-points encountered in the stack.
- Perform a *vertical* splitting of the choice-points;

Additionally, observe that the splitting strategy adopted (e.g., horizontal splitting, vertical splitting) can be changed depending on the specific structure of the computation. For example, along these lines Silva et al. [34] have recently proposed a splitting strategy—*diagonal splitting*—that combines vertical and horizontal splitting and performs well for certain classes of benchmarks.

6.5 Stack-splitting and And-Parallelism

Or-parallelism is typically only one of the forms of parallelism that one can exploit from logic programming and other search-based systems. The development of a single branch of the search tree typically requires a large number of operations, that could themselves be executed in parallel, leading to what is typically indicated as *And-parallelism.*

The advantages of stack-splitting for realizing and-parallelism are similar to those for or-parallelism. When a processor steals work, it gets a large grain of work. Coupled with well-known run-time optimizations, such as the various parallel versions of the last call optimization (e.g., the *Last Parallel Call Optimization* used in logic programming [17]) which increases the size of the parallel conjunctions, larger grain-sized work can be obtained. Also, if additional processors are not available, optimizations [17] can keep the and-parallel computation as close to sequential execution as possible—just as in or-parallelism, where choice-point can be explored via backtracking.

The stack-splitting technique is expected to be reasonably effective for distributed implementations of and-parallelism. Copying of stacks may seem to be an inordinate amount of effort for and-parallel implementations, but given that implementations such as MUSE are quite efficient, a stack-copying based and-parallel system should also be efficient. In addition, the overhead in setting up the bindings while *joining* the multiple and-parallel threads, so that the continuation goal of the parallel conjunction can be correctly executed, may be quite high; however, we hope that the use of and-trails will keep it to a minimum. Furthermore, the issue of joining will arise in any distributed implementation of and-parallelism, where bindings of commonly accessible variables have to be exchanged between processors to allow for a coherent execution. The use of and-trail is similar, but more efficient, than existing techniques such as *environment closing* [10].

Note that just as optimizations such as the Last Alternative Optimization increase the applicability and the benefits of stack-splitting in the case of or-parallelism, so will optimizations such as the Last Parallel Call Optimization lead to increased applicability of and benefit from stack-splitting in the case of and-parallelism.

7 Supporting Prolog's Sequential Semantics

7.1 Introduction

A parallel Prolog system that maintains Prolog semantics reproduces the behavior of a sequential system (same solutions, in the same order, and with the same termination properties). Sequential Prolog systems include features that allow the programmer to introduce a component of sequentiality in the execution. These may be in the form of facilities to express side-effects (e.g., I/O) or constructs to control the order of construction of the computation (e.g., pruning operations, user-defined search strategies). In a parallel system, such *Order Sensitive Components ($\mathcal{OSC}$)* need to be performed in the *same order as in a sequential execution*; if this requirement is not met, the parallel computation may reproduce a semantics different from the one indicated by the programmer [19].

In the context of Prolog, there are three different classes of $\mathcal{OSC}$: *side-effects* predicates (e.g., I/O), *meta-logical* predicates (e.g., test the instantiation state of variables), and *control* predicates (e.g., for pruning branches of the search tree). In the context of or-parallelism only certain classes of $\mathcal{OSC}$ require sequentialization across parallel computations—only side-effects and control predicates. The presence of $\mathcal{OSC}$ does not require a sequentialization of the whole execution involved, only the $\mathcal{OSC}$ themselves need to be sequentialized. If the $\mathcal{OSC}$ are infrequent and spaced apart, good speedups can be obtained, even in a DMP. The correct order of execution of $\mathcal{OSC}$ corresponds to an in-order traversal of the computation tree. A specific $\mathcal{OSC}$ α can be executed only if all the $\mathcal{OSC}$ that precede α in the traversal have been completed. Detecting when all the $\mathcal{OSC}$ to the left have been executed is an undecidable problem, thus requiring the use of approximations. The most commonly used approximation is to execute an $\mathcal{OSC}$ only when the branch containing it becomes the left-most branch in the tree [20]. Thus, we approximate the termination of the preceding $\mathcal{OSC}$ by verifying the termination of the *branches* that contain them. Most of the schemes proposed [29, 19] rely on traversals of the tree, where the computation attempting an $\mathcal{OSC}$ walks up its branch verifying the termination of all the branches to its left. These approaches can be realized [20, 1, 35] in presence of a shared representation of the computation tree—required to check the status of other executions without communication. These solutions do not scale to the case of DMP, where a shared representation of the computation tree is not available. Simulation of a shared representation is infeasible, as it leads to unacceptable bottlenecks [36]. Some attempts to generalize mechanisms to handle $\mathcal{OSC}$ to DMPs have been made [2], but only at the cost of sub-optimal scheduling mechanisms. It is unavoidable to introduce a communication component to handle $\mathcal{OSC}$ in a distributed setting. We demonstrate that stack-splitting can be modified to solve this problem with minimal communication [39]. The modification is inspired by the optimal algorithms for $\mathcal{OSC}$ studied in [29].

7.2 Optimal Algorithms for Order-sensitive Executions

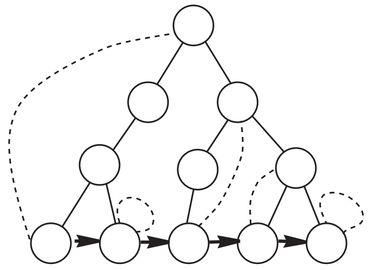


Figure 15: Structures for Order-Sensitive Computations

The problem of efficiently handling $\mathcal{OSC}$ during parallel executions has been pragmatically tackled in a variety of proposals [19]. Nevertheless, only recently the problem has been formally studied, deriving solid theoretical foundations regarding the inherent complexity of testing for leftmostness in a dynamically changing tree [29]. Let $\mathcal{T} = \langle N, E\rangle$ be the computational tree (where N are its nodes and E the current edges). The computation tree is dynamic; the modifications to the tree can be described by two operations: expand which adds a (bounded) number of children to a leaf, and delete which removes a leaf from the tree. Whenever a branch encounters a side-effect, it must check if it can execute it. This check boils down to verifying that the branch containing the side-effect is currently the leftmost active computation in the tree. If n is the current leaf of the branch where the side-effect is encountered, its computation is allowed to continue only if $\mu(n) = \mathsf{root}$, where $\mu(n)$ indicates the highest node m in the tree (i.e., closest to the root) such that n is in the leftmost branch of the subtree rooted at m. $\mu(n)$ is also known in the parallel logic programming community as the *subroot node* of n [20]. Thus, checking if a side-effect can be executed requires the ability of performing the operation find_subroot(n) which, given a leaf n, computes the node $\mu(n)$.

From the data structures studied in [29], it is possible to derive the following: any sequence of expand, delete, and find_subroot operations can be performed in $O(1)$ time per operation on pure pointer machines—i.e., without the need of complex arithmetic (i.e., the solution does not rely on the use of "large" labels). The data structure used to support this optimal solution is based on maintaining a dynamic list which represents the frontier of the tree (Fig. 15). The dynamic list can be updated in $O(1)$ time each time leaves are added or removed (i.e., when expanding a branch and performing backtracking). Subroot nodes can be efficiently maintained for each leaf—in particular, each delete operation affects the subroot node of at most one other leaf. Identification of the computations an $\mathcal{OSC}$ α depends on can be simply accomplished by traversing the list of leaves right-to-left from α. Executability (i.e., leftmostness) can be verified in constant time by simply checking whether the subroot of the leaf points to the root of the tree. This solution is feasible in a shared memory context but requires adjustment in a distributed memory context. In the rest of this section we show how stack-splitting can incorporate a good solution to the problem, following the spirit of this optimal scheme.

7.3 Stack-Splitting and Order-sensitive Computations

Determining the executability of an $\mathcal{OSC}$ α in a distributed memory setting requires two coordinated activities: *(a)* determining *what are* the computations to the left of α in the computation tree—i.e., who are the processors that have acquired work in branches to the left of α; *(b)* determining what is the *status* of the computations to the left of α. On DMPs, both steps require exchange of messages between processors. The main difficulty is represented by step *(a)*—without the help of a shared data structure, discovering the position of the different processors requires arbitrary localization messages exchanged between the processor in charge of α and all the other processors. What we propose is a shift in perspective, directed from the ideas presented in Section 7.2: through a simple modification in the strategy for stack-splitting, we can guarantee that processors are aware of the position of their subroot nodes. Thus, instead of having to locate the subroot nodes whenever an $\mathcal{OSC}$ occurs, these are implicitly located (without added communication) whenever a sharing operation is performed (a very infrequent operation, compared to the frequency of $\mathcal{OSC}$ steps). Knowledge of the position of the subroot nodes allows processors to maintain an approximation of the ordering of the leaves of the tree, which in turn can be used to support the execution of step *(b)* above.

In the original stack-splitting procedure—using vertical splitting (Section 3.2)—during a sharing operation the parallel choice-points are alternatively split between two processors. The processor that is giving the work keeps the bottom-most choice-point, the third bottom-most choice-point, the fifth bottom-most choice-point, etc. The processor that receives the work keeps the second bottom-most choice-point, the fourth bottom-most choice-point, etc. In our previous works [18, 37] we have demonstrated that this splitting strategy is effective and leads to good speedups for large classes of representative benchmarks. The alternation in the distribution of choice-points is aimed at reducing the danger of focusing a particular processor on a set of fine-grained computations. This strategy for splitting a computation branch between two processors has a significant drawback w.r.t. execution of $\mathcal{OSC}$, since the two processors, through backtracking, may arbitrarily move left or right of each other. This makes it impossible to know a-priori whether one processor affects the position of the subroot node of other processors, preventing the detection of the position of processors in the frontier of the tree. From Sect. 7.2 we learn that a processor operating on a leaf of the computation tree can affect other processors' subroot nodes only in a limited fashion. The idea can be easily generalized: if a processor limits its activities to the bottom part of a branch, then the number of leaves affected by the processor is limited and well-defined. This observation leads to a modified splitting strategy, where the processor giving work keeps the lower segment of its branch as private, while the processor receiving work obtains the upper segment of the branch. This modification guarantees that the processor receiving work will be always to the right of the processor giving the work. Since the result of a sharing operation is always broadcast to all the processors—to allow processors to maintain an approximate view of the distribution of work—this method also allows each processor to have an approximate view of the composition of the frontier of the computation tree. The next sections show how this new splitting strategy can be made effective to support $\mathcal{OSC}$ without losing parallel performance.

7.3.1 Implementation

Data Structures: In order to support the new splitting strategy and use it to support $\mathcal{OSC}$ steps, each processor will require only two additional data structures: *(1)* the *Linear Vector* and *(2)* the *Waiting Queue*. Each processor keeps and updates a *linear vector* which consists of an array of processor Ids that represents the linear ordering of the processors in the search tree—i.e., the respective position of the processors within the frontier of the computation tree (section 7.2). The idea behind this *linear* *vector* is that whenever a processor wants to execute an $\mathcal{OSC}$, it first waits until there are no processors Ids to its left on the *linear vector*. Such a status indicates that all the processors that were operating to the left have completed their tasks and moved to the right side of the computation tree, and the subroot node has been pushed all the way to the root of the tree. Once this happens, the processor can safely execute the $\mathcal{OSC}$, being left-most in the search tree. Initially, the linear vector of all processors contains only the Id of the first running processor. In the original bottom-most scheduler developed for stack-splitting (Section 4), every time a sharing operation is performed, a `Send_LoadInfo` message is broadcast to all processors; this is used to inform all processors of the change in the workload and of the processors involved in the sharing. For every `Send_LoadInfo` message, each processor updates its linear vector by moving the Id of the processor that received work immediately to the right of the Id of the processor giving work. Each processor also maintains a *waiting queue* of Ids, representing all the processors that are waiting to execute an $\mathcal{OSC}$ but are located to the right of this processor. Whenever a processor enters the *scheduling* state to ask for work, it informs all processors in its waiting queue that they no longer need to wait on it to execute their $\mathcal{OSC}$.

The Procedure: In stack-splitting (Section 4), a processor can only be in one of two states: *running state* or *scheduling state*. In order to handle $\mathcal{OSC}$, we need another state: the *order-sensitive* state. All processors wanting to execute an $\mathcal{OSC}$ will enter this state until it is safe for them to execute their $\mathcal{OSC}$. The transition between the states requires the introduction of three types of messages: (1) `Request_OSC`, (2) `OSC_Acknowledgment`, and (3) `Reply_In_OSC`. A `Request_OSC` will be used by a processor that wants to execute an $\mathcal{OSC}$, and sent to all processors to its left in its linear vector. When a processor receives a message of this type, and it is not located to the left of the sending processor, a reply message with tag `OSC_Acknowledgment` will be sent back. This message informs the $\mathcal{OSC}$ processor that it no longer needs to wait on this processor. When a processor is in running state and gives work to another processor, a `Send_LoadInfo` message is sent to all other processors informing them of the new load information. Each processor receiving the `Send_LoadInfo` message updates its own linear vector by placing the Id of processor that received the work to the right of the processor giving work. When a processor is in scheduling state and receives work from another processor p, it will also update its linear vector by placing its Id to the immediate right of p. Once a processor arrives to the order-sensitive state, it first sends a `Request_OSC` to all the processors to its left in its linear vector. It then waits for an `OSC_Acknowledgment` message from each of them. An `OSC_Acknowledgment` is sent by a processor when it is no longer to the left of the processor wanting to execute the $\mathcal{OSC}$. When this message is received, the Id of the processor sending it will be removed from the linear vector. Notice that when the processor is waiting for these messages, it may receive `Send_LoadInfo` messages. If this happens, the processor has to update its linear vector. In particular, if due to this sharing operation a processor moves to its left, a `Request_OSC` message needs to be sent to this processor as well. Once the processor receives an `OSC_Acknowledgment` from all these processors, it can safely perform the $\mathcal{OSC}$. Processors in the order-sensitive state are not allowed to share work; requests to share work are denied with the `Reply_In_OSC` message. When a processor is in running state and receives a `Request_OSC` message, it consults its linear vector and reacts in the following way. If the Id of the processor wanting to execute an $\mathcal{OSC}$ is to its right in the linear vector, the Id of the requesting processor is inserted in the waiting queue. When the running processor runs out of work and moves to the scheduling state, an `OSC_Acknowledgment` message will be sent back to the processor wanting to execute the $\mathcal{OSC}$. If the Id of the processor wanting to execute the $\mathcal{OSC}$ is to its left, an `OSC_Acknowledgment` message is immediately sent back to the processor wanting to execute the $\mathcal{OSC}$. This means that the running processor is no longer to the left of the processor wanting to execute the $\mathcal{OSC}$. When a processor enters the scheduling state, it dequeues all the Ids from its waiting queue and sends an `OSC_Acknowledgment` to all these processors, informing them that it is no longer to their left. Similarly, the linear vector of a processor in any of the states that receives a `Request_Work` and cannot give away work can also be easily updated. In this case, the Id of the processor requesting work can be safely removed from the linear vector. Just as we attach load information to messages in the traditional stack-splitting scheduling algorithm, we also attach updated load information to these three new messages.

7.3.2 Implementation Details

Partitioning Ratios: The stack-splitting modification divides the stack of parallel choice-points into two contiguous partitions, where the bottom partition is kept by the processor giving work and the upper partition is given away. This stack-splitting modification guarantees that the processor that receives work will be to the immediate right of the other processor. The question is what is the partitioning ratio that will produce the best results? We first tried using a partition where the processor that is giving work keeps the bottom half of the branch and only gives away the top half. After experimenting with lots of different partition ratios, we found out that with a partition ratio of $3/4 - 1/4$ where the processor that is giving work keeps the bottom 3/4 of the parallel choice-points and gives away the top 1/4 of the parallel choice-points, our benchmarks without side-effects obtain excellent speedups—similar to our original alternating splitting [37]. When we run our benchmarks with side-effects, the partition ratio of $3/4 - 1/4$ performed superior to the partition ratio of 1/2. The explanation is that it is better for processors to get smaller pieces of work with fewer side-effects than large pieces with lots of side-effects.

Messages Out of Order: `Send_LoadInfo` messages may arrive out of order and then the linear vectors may be outdated. E.g., processor 2 receives from processor 0 a `Request_Work` message but decides not to share work. Since processor 0 is requesting work, processor 2 removes 0 from its linear vector. Later on, processor 0 gets work from processor 1, and processor 1 broadcasts a `Send_LoadInfo` message. Afterwards, processor 0 gives work to processor 3 and also broadcasts a `Send_LoadInfo` message. Now, suppose that processor 2 receives the second `Send_LoadInfo` message first and the first `Send_LoadInfo` next. When processor 2 tries to insert 3 to the immediate right of 0 in the linear vector, 0 is not located and therefore 3 cannot be inserted. MPI (used in our system for processor communication) does not guarantee that two messages sent from different processors at different times will arrive in the order that they were sent. The scenario presented above can be avoided if, in every sharing operation, both the involved processors broadcast a `Send_LoadInfo` message to all the other processors. In this case every processor will be informed that a sharing operation occurred either by the giver or by the receiver of work. Processor 2 in the above scenario will first know that processor 0 obtained work from processor 1, and then will know that processor 0 gave work to processor 3. Duplication of `Send_LoadInfo` messages is handled through the use of two dimensional arrays $send1$ and $send2$ of size N^2, where N is the total number of processors; $send1[i][j]$ ($send2[i][j]$) is incremented when a sharing message from i to j is received from processor i (j). The linear vector will be updated only if $send1[i][j] > send2[i][j]$ ($send2[i][j] > send1[i][j]$) and the message comes from i (j).

7.4 Discussion

The modified stack-splitting strategy described in this section provides a strategy to bias the exploration of the branches in the search tree in the left-to-right order. As far as exploring the search space with a bias towards exploring the branches to the left is concerned, it will depend on the choice-point splitting strategy used. Consider the choice-point with alternatives `a1` through `a5` shown in Fig. 16(i). Two possible splittings are shown in Fig. 16(ii) and 16(iii). In the first one (Fig. 16(ii)), the list of alternatives is split in the middle: processor P1 will be working on the left half of the tree rooted at this choice-point a, processor P2 on the right half. In contrast, in Fig. 16(iii), the untried alternatives are distributed alternately between the two choice-points. This splitting strategy is more likely to produce a search that is biased to the left. This suggests that modifications similar to the one presented in this work can be extended to the case of horizontal splitting.

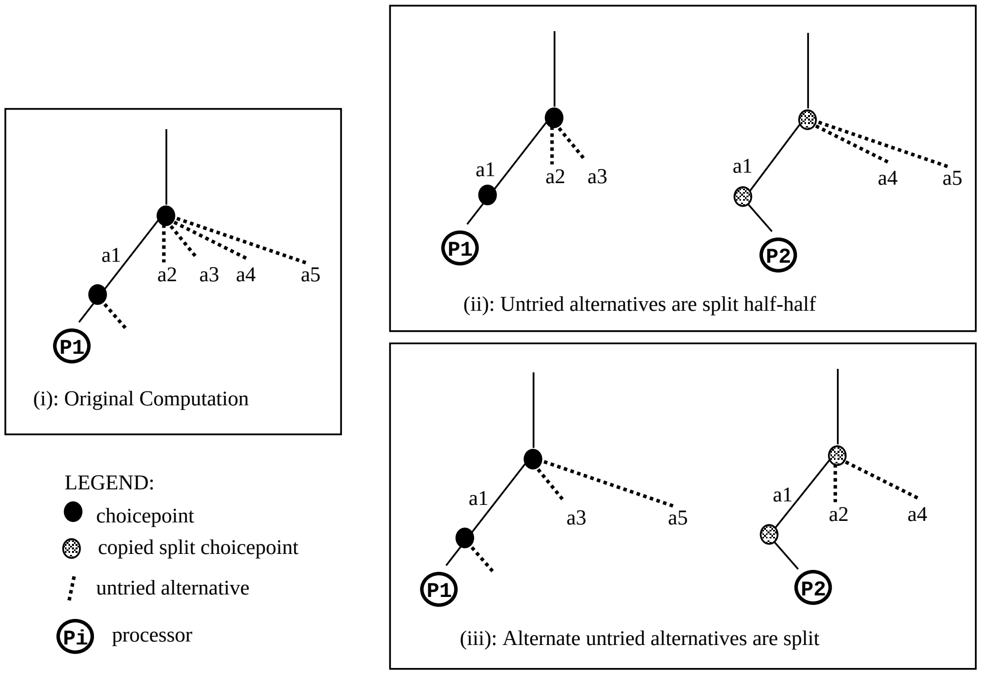

a1
a2
a3
a4
a5
P1
(i): Original Computation
a1
a2
a3
P1
a1
a4
a5
P2
(ii): Untried alternatives are split half-half
a1
a3
a5
P1
a1
a2
a4
P2
(iii): Alternate untried alternatives are split
LEGEND:
choicepoint
copied split choicepoint
untried alternative
Pi
processor

Figure 16: Distribution of Unexplored Alternatives

8 Performance Results

8.1 Shared Memory Implementation

The stack-splitting procedure has been implemented on top of the commercial ALS Prolog system using the MPI library for message passing. The whole system runs on a Sun Enterprise 4500 with fourteen processors. While the Sun Enterprise is an SMP, it should be noted that all communication—during scheduling, copying, splitting, etc.— is done using messages. This has done to enable an easy migration of the system to a truly distributed Beowulf machine (a network of Pentium II nodes connected by a Myrinet Switch). The timing results in seconds from our incremental stack-splitting system on the 14 processor Sun enterprise are presented in Table 1. This system is based on the scheduling strategy described above.

The benchmarks that we have used to test our system are the following. The *9 Costas* and *8 Costas* benchmarks compute the Costas sequences of length 9 and 8 respectively. The *Knight* benchmark consists of finding a path of knight-moves on a chessboard of size 5, starting at (1,1) and finishing at (1,5), and visiting every square on the board just once. The *Stable* benchmark is a simple engine to compute the models of a logic program with negation. The *Send More* benchmark consists of solving the classical crypto-arithmetic puzzle. The *8 Puzzle* benchmark is a solution to the puzzle involving a

3-by-3 board with 8 numbered tiles. The *Bart* benchmark is a simulator used to test the safety of the controller for a train. The *Solitaire* benchmark is a solution to the standard game involving a triangular board with pegs and one empty hole. The *10 Queens* and *8 Queens* benchmarks consist of placing a number of queens on a chessboard so that no two queens attack each other. The *Hamilton* benchmark consists of finding a closed path through a graph such that all the nodes of the graph are visited once. The *Map Coloring* benchmark consists of coloring a planar map.

The *9 Costas*,*8 Costas*, *Knight*, *Stable*, *10 Queens*, *8 Queens*, *Hamilton*, and *Map Coloring* benchmarks compute all the possible solutions. The *Send More*, *8 Puzzle*, *Bart*, and *Solitaire* benchmarks compute only one solution. The implementation of the *9 Costas*, *8 Costas*, and *Bart* benchmarks is fairly complicated. The implementation of the rest of the benchmarks is fairly simple. However, all benchmarks provide sufficiently different program structures to extensively test the behavior of the parallel engine.

Benchmark	**# Agents**				
	1	**2**	**4**	**8**	**14**
9-Costas	715.369	368.298	184.141	92.165	53.453
Stable	653.705	368.943	185.474	92.811	53.860
Knight	275.737	141.213	70.528	35.539	22.403
Send More	115.183	65.271	31.447	16.496	9.686
8-Costas	66.392	34.281	17.192	8.680	5.202
8-Puzzle	52.945	29.601	15.026	7.845	4.754
Bart	25.562	15.411	6.868	3.577	2.144
Solitaire	12.912	7.598	3.813	2.029	1.335
10-Queens	7.575	3.922	2.087	1.378	1.141
Hamilton	6.895	3.879	1.940	1.151	0.761
Map Coloring	2.036	1.298	0.696	0.479	0.430
8-Queens	0.306	0.198	0.143	0.157	0.149

Table 1: Incremental Stack-splitting (sec.)

We observe that for benchmarks with substantial running time (9-Costas, 8-Costas, Knight, and Stable) the speed-ups are very good. We also observe that for benchmarks with not so substantial but also not very small running time (Send More, 8-Puzzle, Bart, Solitaire, and Hamilton) the speed-ups are still quite good. Note that for the benchmarks with small running time (10-Queens, Map Coloring and 8-Queens) the speed-ups deteriorate. See Fig. 17 under the label *Incremental.*

The observations made above are consistent with our belief that parallel systems should be used for parallelizing programs with somewhat substantial running times. For programs with small-running times, there is not enough work to offset the cost of exploiting parallelism. Nevertheless, our system is reasonably efficient, given that even for small benchmarks it can produce reasonable speed-ups.

In order to compare our incremental stack-splitting system we have also implemented two other techniques using *non-incremental stack-copying*: we copy the entire WAM data areas when sharing work instead of copying them incrementally as described above. One of these techniques is based on stack-splitting, and the other is based on *scheduling on top-most choice-point*: this methodology transfers between agents only the highest choice-point (i.e., closer to the root) in the computation tree which contains unexplored alternatives. The timing results in seconds from these other systems are presented in Tables 2 and 3. These two systems also used the scheduling strategy described above. The speed-ups for these systems are shown in Fig. 17 under the labels *Complete* and *Top.*

Most benchmarks show that the incremental stack-splitting system obtains higher speed-ups than the non-incremental systems. Between the non-incremental systems, the stack-splitting system performs better in most of the benchmarks than the scheduling on top-most choice-point system. This is particularly evident in the case of the Hamilton benchmark (Fig. 17). Some of the benchmarks (9-Costas, 8-Costas, and Knight) show almost no difference in performance among the three systems. One of the reasons why this is happening is that during the execution of these benchmarks there must be only one or two parallel choice-points which are given away or split per sharing. Analyzing the source code for these benchmarks, we see that in the three benchmarks just one parallel choice-point contains all the parallel work.

Finally, the incremental stack-splitting system introduces a reasonably small overhead with respect to the original sequential ALS Prolog system. Our PALS system, on a single agent, is on average 5% slower than the sequential ALS system.

Benchmark	**# Agents**				
	1	**2**	**4**	**8**	**14**
9-Costas	715.963	366.385	182.654	93.602	52.901
Stable	614.582	374.259	184.404	93.884	54.022
Knight	276.849	141.118	70.568	35.741	20.958
Send More	116.518	65.936	31.892	16.882	10.364
8-Costas	66.221	34.053	17.126	8.656	5.202
8-Puzzle	52.909	29.615	15.148	8.206	5.654
Bart	25.734	13.898	6.863	3.704	2.382
Solitaire	12.676	7.552	3.910	2.177	1.606
10-Queens	7.557	3.935	2.116	1.483	1.535
Hamilton	6.908	3.910	1.963	1.284	0.991
Map Coloring	2.009	1.332	0.721	0.476	0.675
8-Queens	0.308	0.194	0.158	0.161	0.138

Table 2: Timings for Complete Stack-splitting (sec.)

Benchmark	**# Agents**				
	1	**2**	**4**	**8**	**14**
9-Costas	756.785	385.251	192.157	96.560	55.602
Stable	644.989	384.961	192.991	99.071	55.764
Knight	270.672	139.307	69.951	35.338	22.504
Send More	111.345	64.650	32.562	16.504	9.806
8-Costas	70.362	35.899	19.383	9.197	5.441
8-Puzzle	53.843	48.754	15.490	12.731	8.111
Bart	26.419	14.378	7.513	3.870	2.540
Solitaire	11.883	7.187	3.664	1.955	1.363
10-Queens	7.595	3.857	2.117	1.330	1.160
Hamilton	6.964	4.061	2.246	1.941	1.606
Map Coloring	2.207	1.389	0.816	0.595	0.469
8-Queens	0.304	0.194	0.181	0.155	0.177

Table 3: Top-most Scheduling (sec.)

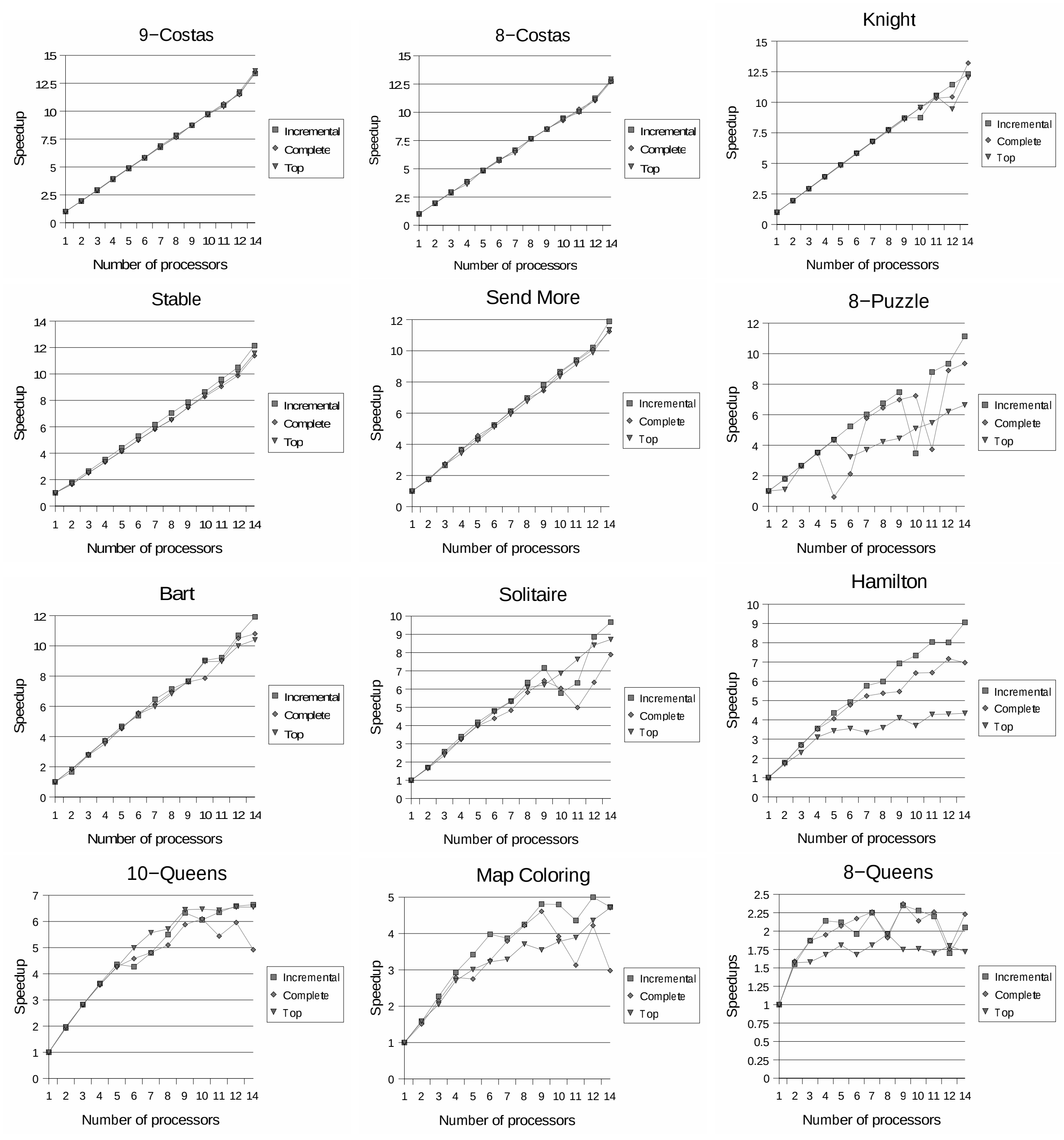


Figure 17: Speed-ups

8.2 Beowulf Implementation

8.2.1 Stack-Splitting

The stack-splitting procedure has been implemented by modifying the commercial ALS Prolog system, using the MPI library for message passing. The whole system runs on a truly distributed machine (a network of 32 Pentium II nodes connected by Myrinet-SAN Switches). All communication—during scheduling, copying, splitting, etc.— is done using explicit message passing via MPI.

The timing results in seconds from our incremental stack-splitting system are presented in Table 4. The modifications made to the ALS WAM are very localized and reduced to the minimum necessary. This has allowed us to keep a very clean design—that, we hope, can be easily ported to other WAM-based implementations—and to keep under control the parallel overhead—our engine running on a single processor is on average only 10% slower than the ALS WAM. The corresponding speed-ups are presented in Fig. 18. under the label *incremental*.

Benchmark	**# Processors**					
	1	**2**	**4**	**8**	**16**	**32**
9 Costas	412.579	210.228	105.132	52.686	26.547	14.075
Knight	159.950	81.615	40.929	20.754	10.939	8.248
Stable	62.638	35.299	17.899	9.117	4.844	3.315
Send More	61.817	32.953	17.317	8.931	4.923	3.916
8 Costas	38.681	19.746	9.930	5.052	2.733	1.753
8 Puzzle	27.810	15.387	8.442	10.522	3.128	5.940
Bart	13.619	7.958	4.090	2.031	1.600	0.811
Solitaire	5.909	3.538	1.811	1.003	0.628	0.535
10 Queens	4.572	2.418	1.380	0.821	1.043	0.905
Hamilton	3.175	1.807	0.952	0.610	0.458	0.486
Map Coloring	1.113	0.702	0.430	0.319	0.318	0.348
8 Queens	0.185	0.162	0.166	0.208	0.169	0.180

Table 4: Timings for Incremental Stack-Splitting (Time in sec.)

We observe that for benchmarks with substantial running time (*9-Costas*, *Knight*, *8-Costas*, *Stable*, and *Send More*) the speed-ups are very good. We also observe that for benchmarks with not so substantial but also not very small running time (*Bart*, *Solitaire*, *8-Puzzle*, and *Hamilton*) the speed-ups are still quite good. Note that for the benchmarks with small running time (*10-Queens*, *Map Coloring* and *8-Queens*) the speed-ups deteriorate. This is consistent with our belief that DMP implementations should be used for parallelizing programs with coarse-grained parallelism. For programs with small-running times, there is not enough work to offset the cost of exploiting parallelism using a distributed communication model. Nevertheless, our system is reasonably efficient, given that even for small benchmarks it can produce some speed-ups. It is also interesting to observe that in no cases we have observed slow-downs due to parallel execution—thanks to the simple granularity control mechanisms embedded in the scheduler.

One of the objectives of the experiments performed is to validate the effectiveness of incremental stack-splitting as a methodology for efficient exploitation of parallelism on DMPs. In particular, there are two aspects that we were interested in exploring: *(i)* verifying the effectiveness of stack-splitting versus a more "direct" implementation of the stack-copying method as implemented in MUSE [1] (i.e., keeping single copies of choice-points around the system); *(ii)* verifying the impact of *incremental*

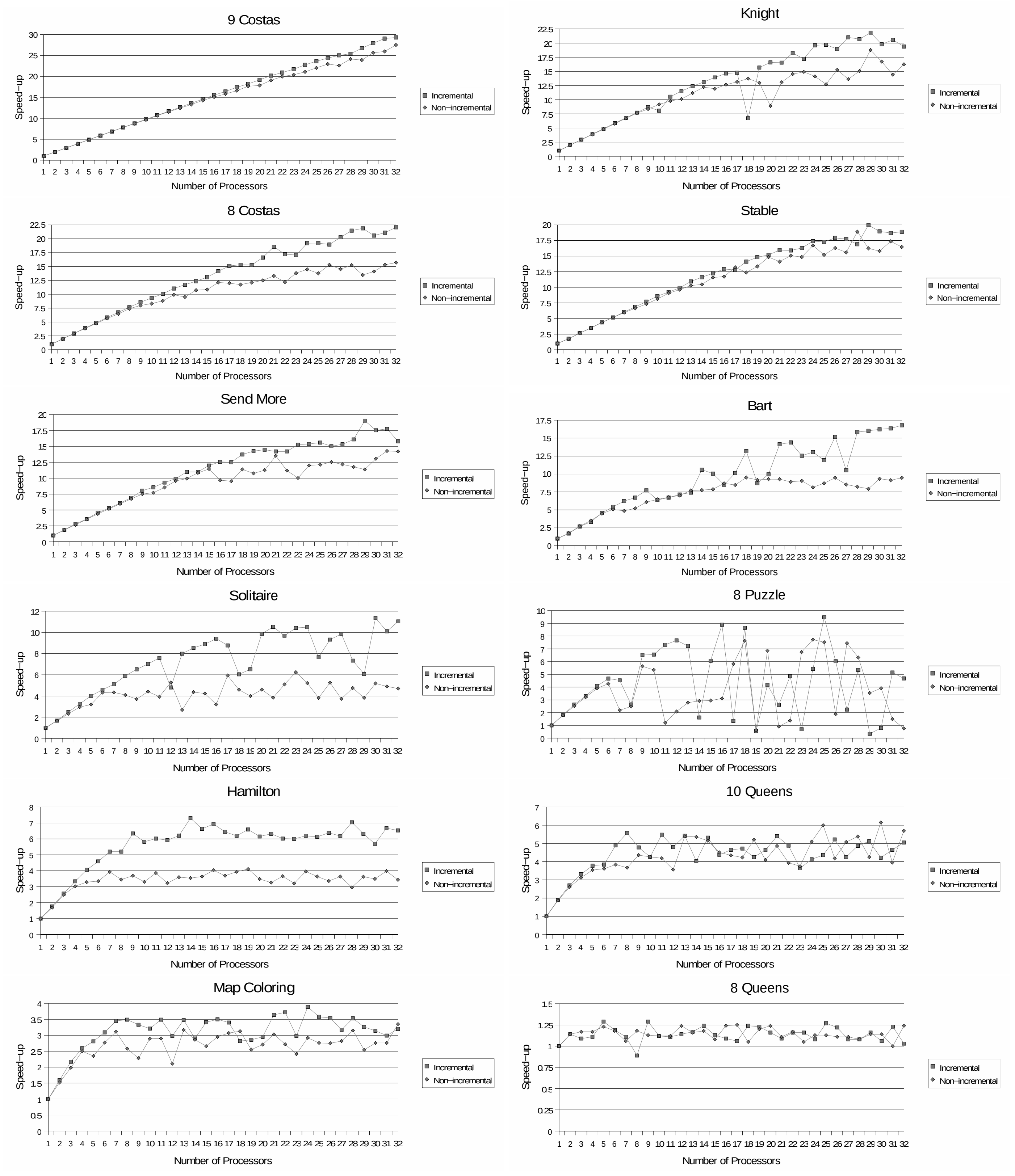


Figure 18: Incremental Stack-Splitting vs. Non-Incremental Stack-Splitting

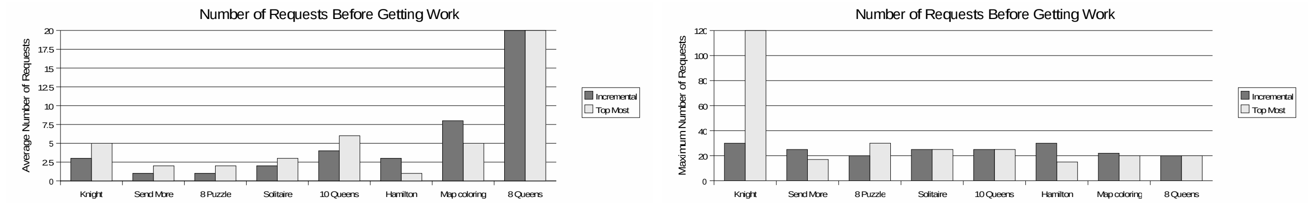


Figure 19: Average and Maximum Number of Tries to Acquire Work

splitting.

Validity of stack-splitting vs. stack-copying can be inferred from the experiments described in Sect. 8.2.2: a direct implementation of stack-copying would produce the same amount of communication traffic as some of the variations of scheduling tested, and thus incur the same kind of problems described in Sect. 8.2.2.

In order to evaluate the impact of incrementality, we have measured the performance of the system without the use of incremental splitting—i.e., each time a sharing operation takes place, a complete copy of the WAM data areas is performed. The results obtained from this experiment are reported in Fig. 18: the figure compares the speed-ups observed with and without incremental splitting. We can observe that our incremental stack-splitting system obtains higher speed-ups than the non-incremental stack-copying system. As expected, the difference is more significant in those benchmarks where a large number of parallel choice-points is generated, as there is an increased possibility of applying incremental splitting. It is also important to observe that in no cases the incremental behavior has lead to a degradation of performance w.r.t. non-incremental splitting.

8.2.2 Scheduling

One of the major reasons to adopt stack-splitting, as described earlier, is the ability to perform scheduling on the bottom-most choice-point. Other DMP implementations of or-parallelism have reversed to the use of scheduling on the top-most choice-point, where during a sharing operation only the oldest choice-point with unexplored alternatives is exchanged between processors. Top-most scheduling will share only one choice-point at the time, thus relieving the engine from the need of controlling access to shared choice-points.

To validate the effectiveness of our claim, we have developed a top-most scheduler for our system and compared its performance with that of the incremental stack-splitting with bottom-most scheduling. Fig. 20 compares the speed-ups observed using the two different schedulers. As we can observe from Fig. 20, in most benchmarks bottom-most scheduling provides a sustained speed-up considerably higher than top-most scheduling. In the remaining benchmarks, top-most and bottom-most scheduling provide similar results, as a small number of choice-points are created and only one at a time is shared between processors. This is due to the reduced number of calls to the scheduler performed during the execution—processors are busy for a longer period of time than using top-most scheduling.

Another aspect of our implementation that we are interested in validating is the performance of the distributed scheduler. As mentioned in Sect. 5, our scheduler is based on keeping in each processor an "approximated" view of the load in each other processor. The risk that this method may encounter is that a processor may have out-of-date information concerning the load in other processors, and as a consequence it may try to request work from idle processors or ignore processors that may have unexplored alternatives. Fig. 19 provides some information concerning the number of attempts that a processor needs to perform before receiving work. The figure on the left measures the average number

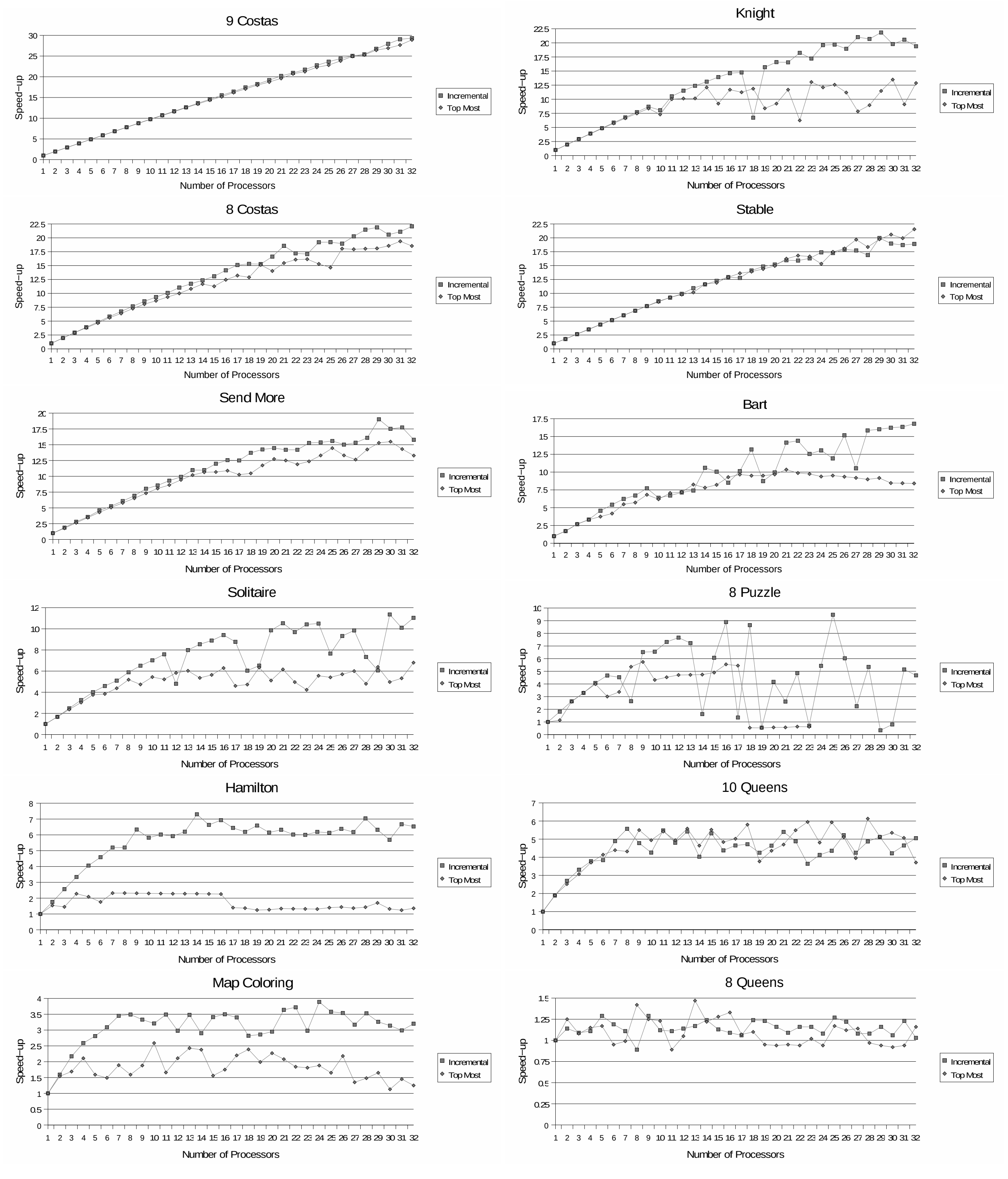


Figure 20: Incremental Stack-Splitting vs. Top Most Scheduling

of requests that a processor has to send; as we can see, the number is very small (1 or 2 requests are typically sufficient) and such number is generally better if we adopt bottom-most scheduling. The figure on the right shows the maximum number of requests observed; these numbers tend to grow towards the end of the computation (when less work is available)—nevertheless, typically only one or two processors achieve these maximum values, while the majority of the processors remain close to the average number of attempts.

To further validate our scheduling approach, we have compared it with an alternative scheduling scheme developed in PALS. This alternative scheme is an implementation of a centralized scheduling algorithm, designed following the guidelines of the scheduler used in the Opera system [7]. In the centralized scheduler approach, only one processor, called *central*, is in charge of keeping track of the load information. Idle processors send their requests for work directly to the central processor. In turn, the central processor is in charge of implementing a matchmaking algorithm between idle and busy processors. The central processor matches requests from idle processors with busy processors with highest load. The central processor is also in charge of detecting termination. When stack-splitting occurs, only the central processor is informed about the load information update. Fig. 21 compares the speed-ups achieved using centralized scheduling with the speed-ups observed using the distributed scheduling approach.[2] As evident from the figure, the speed-ups observed in centralized scheduling are almost negligible—this is due to the inability of the scheduling method to promptly respond to the requests for new work. Also, the use of a reasonably fast interconnection network (Myrinet) leads to the creation of a severe bottleneck at the level of the centralized scheduler. From our experiments we can observe that the centralized scheduler is a feasible solution only if very few coarse-grained tasks are generated. For benchmarks such as *Hamilton*, where a fairly large number of choice-points is generated, the centralized scheduler leads to a considerable loss of performance.

The results presented in [5] suggest that random selection of work may provide a simple and effective alternative when searching for or-parallel work. We have experimented with this idea, by modifying the scheduler to select any busy processor for scheduling instead of the one with the highest load. The idea is to avoid bottleneck situations where multiple idle processors are concentrating their requests for work towards the same busy processor. We have named this new version of the scheduler *Random Scheduler*. In this version, an idle processor searches its load vector for the next processor with load greater than a given small threshold. Fig. 22 compares the speed-ups observed in the Random scheduler with those from the standard bottom-most scheduling with selection of processor with highest load. The results indicate that the Random scheduler is less effective. This suggests that selecting work from the processor with highest load is not a severe bottleneck and sending requests to possibly lightly loaded processors may increase the number of calls to the scheduler.

8.2.3 Tuning the System

The implementation of stack-splitting depends on a number of parameters, such as *(1)* the frequency at which each processor checks for incoming requests, and *(2)* the frequency of propagation of load information. We have performed a number of experiments to study the impact of these parameters on the overall performance.

Regarding the first parameter, the previously presented results make use of a frequency of one test every 200 procedure calls. Fig. 23 and 24 show that this choice was the best, although minimal differences can be observed for different frequency values. Regarding the second parameter, we are

[2]We had to limit the experiments to a smaller number of CPUs due to unavailability of half of the machine at that time.

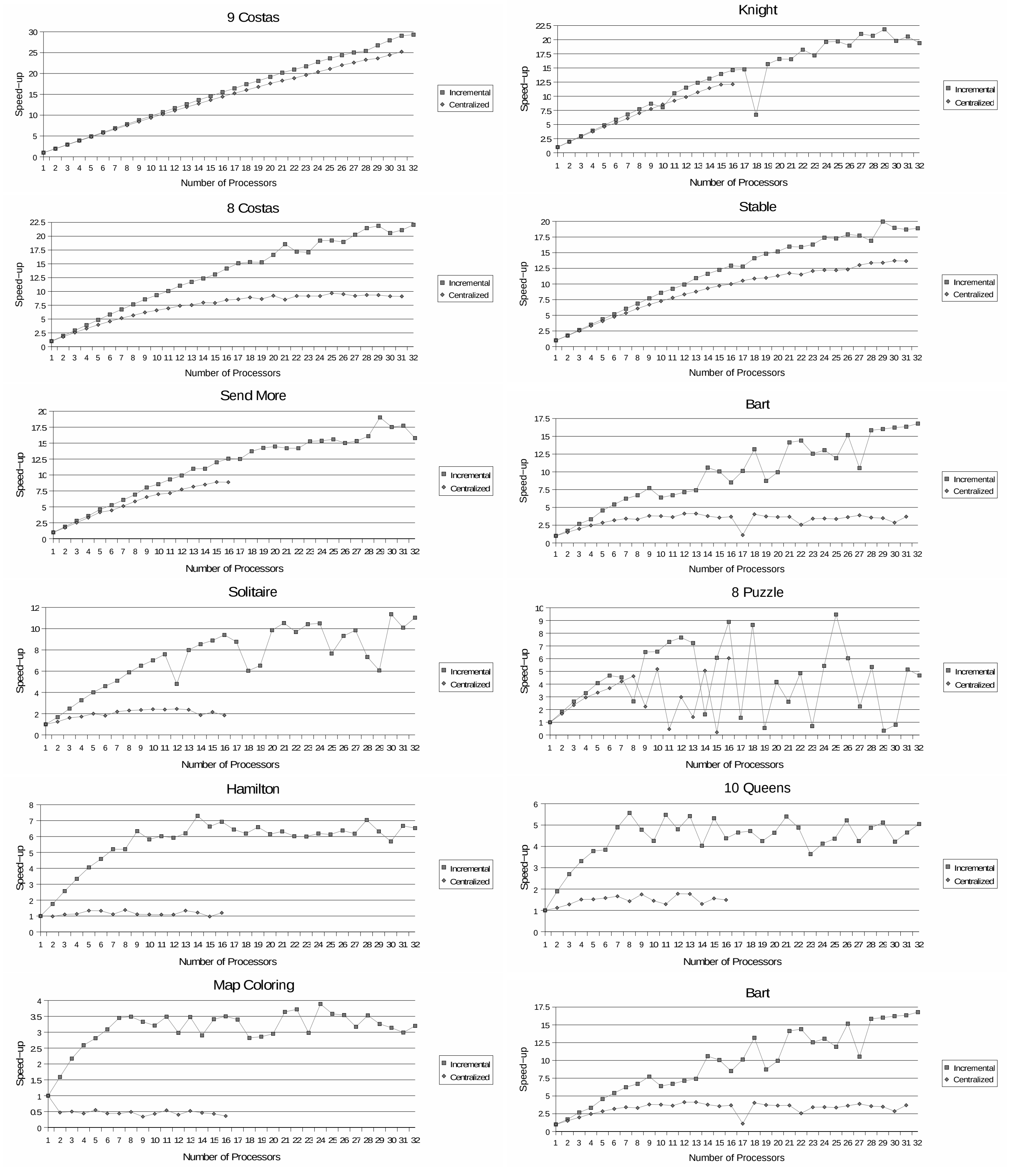


Figure 21: Incremental Stack-Splitting vs. Centralized Scheduling

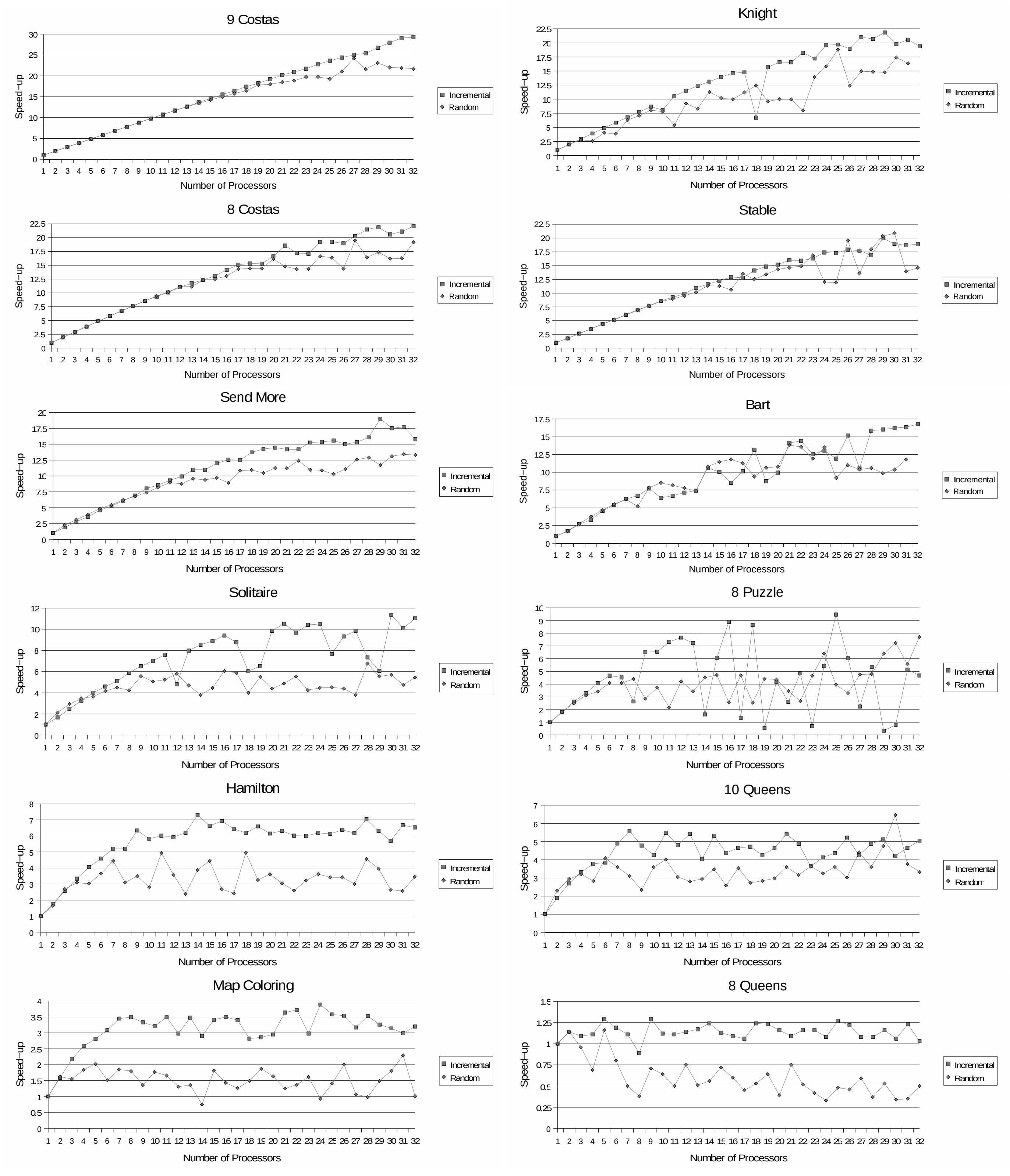


Figure 22: Incremental Stack-Splitting vs. Random Scheduling

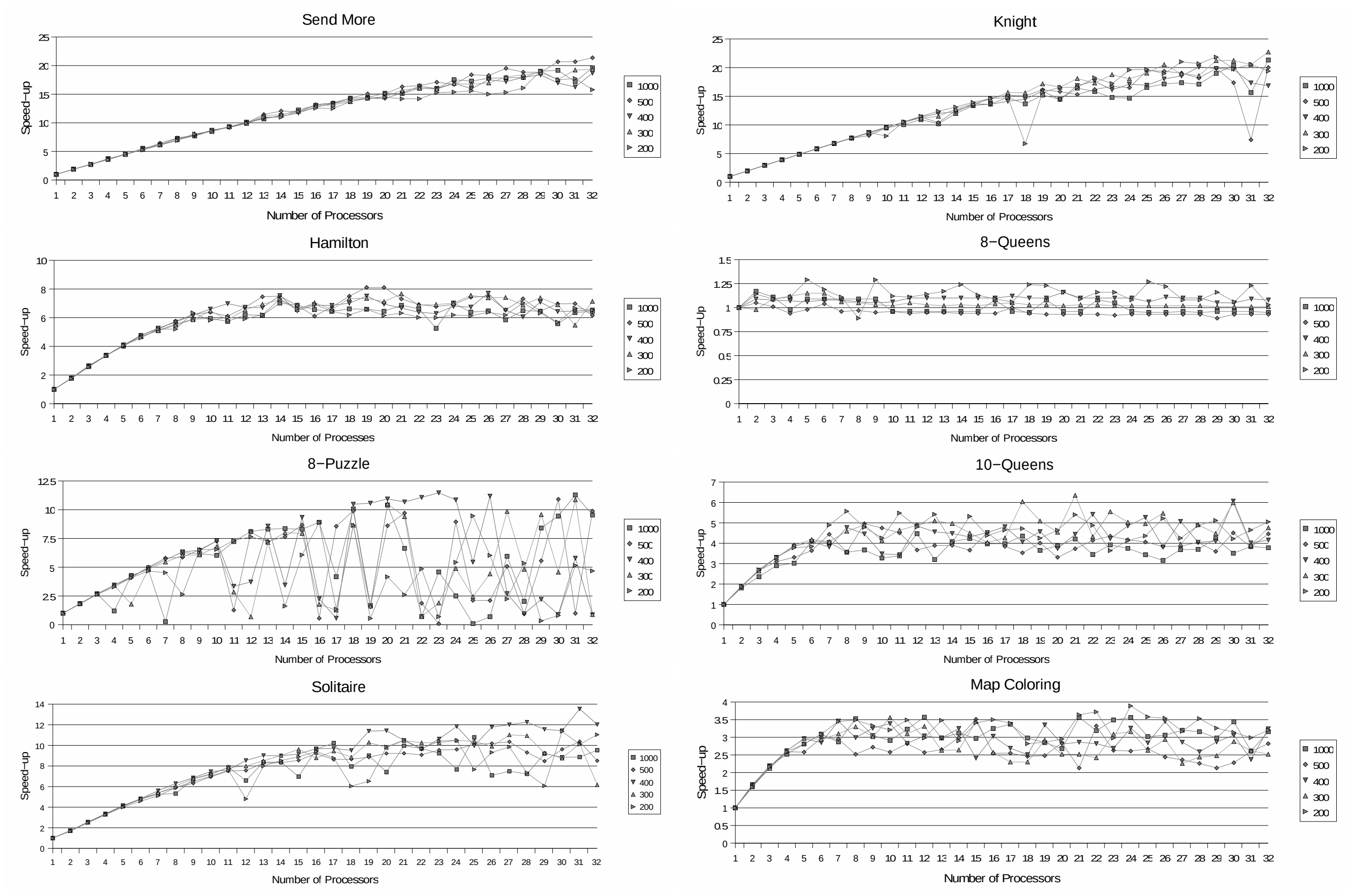


Figure 23: Incremental Stack-Splitting Message Checking Frequencies (1)

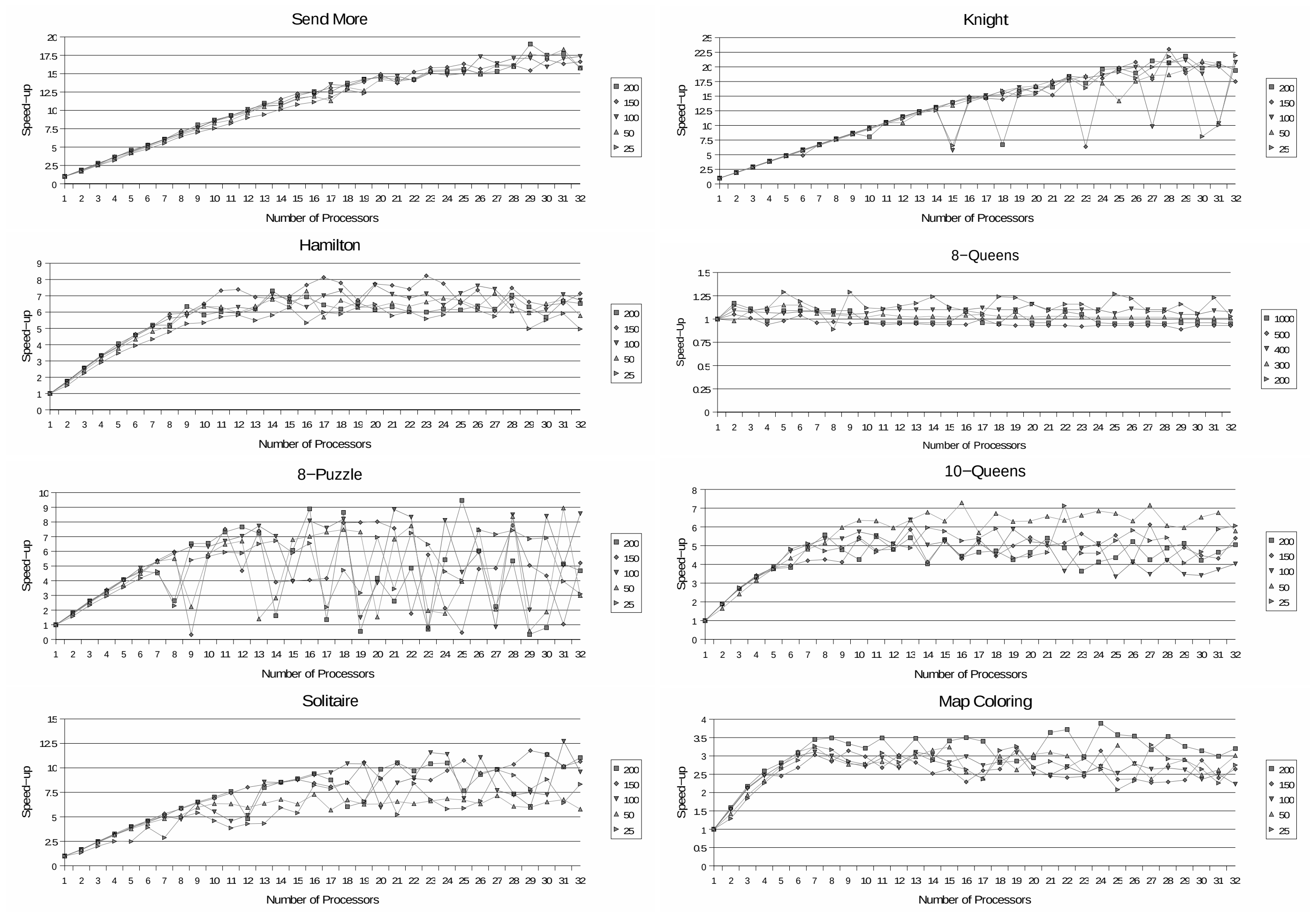


Figure 24: Incremental Stack-Splitting Message Checking Frequencies (2)

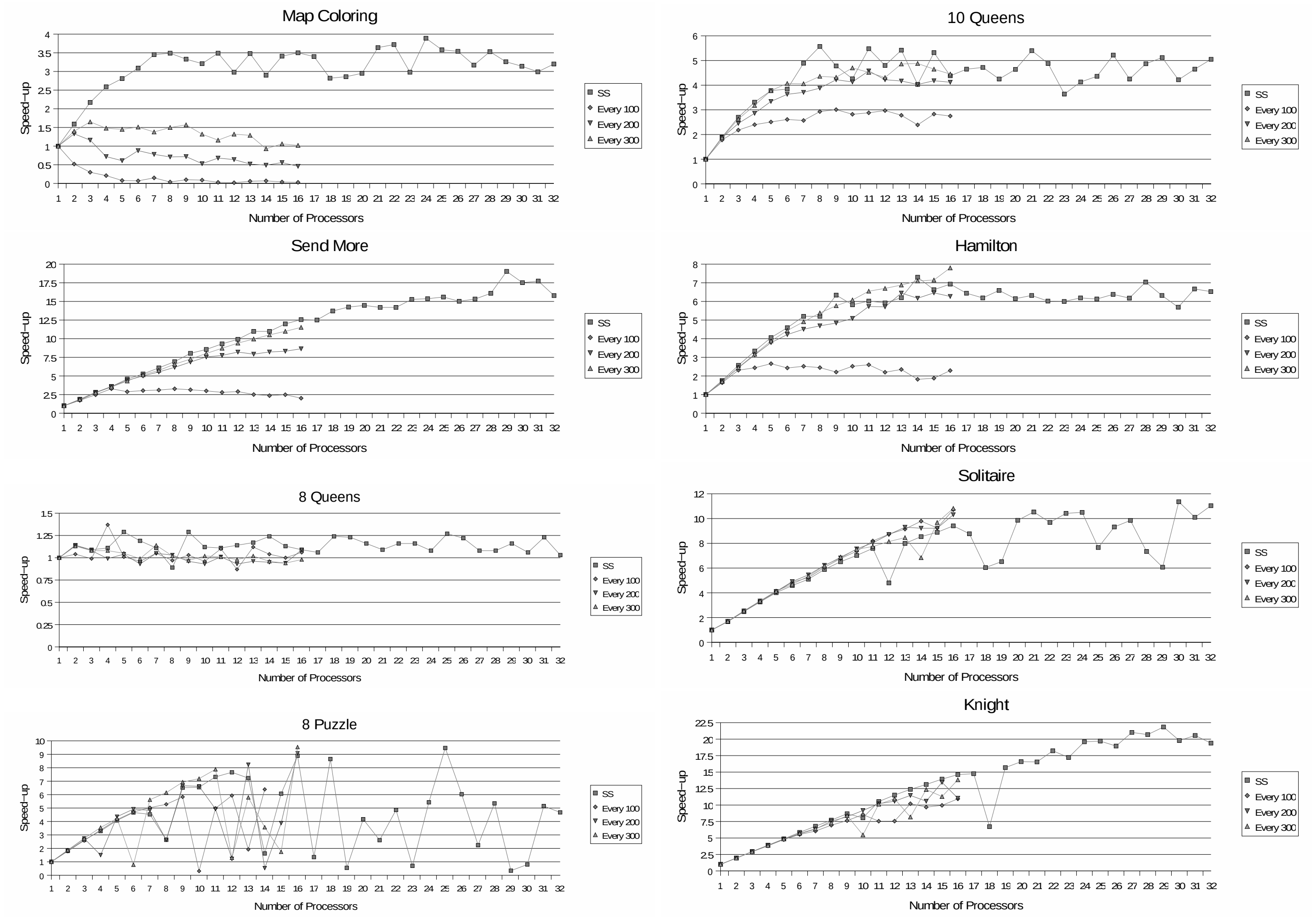


Figure 25: Incremental Stack-Splitting vs. Propagation of Load Information

currently propagating load information only in presence of a sharing operation. We tried to increase the frequency of propagation of load information, hoping to provide processors with a more accurate view of the load in the system. The results from this experiment are reported in Fig. 25. As we can see, with the exception of *Hamilton*, in all other cases increasing the frequency leads only to a higher message traffic without any apparent advantage. In particular, the higher the frequency, the lower is the resulting speed-up.

The last optimization that we tried is based on the termination of our system. In our incremental stack-splitting system, once a processor finds that there is no one to ask for work, it goes into a dead end loop just waiting for the halt signal. Therefore, we modified our system to let an idle process in this situation get out of this dead end loop once it finds that its load vector has been updated so that it can go back to life and ask for work. We call this version *delay termination*. However, we still observed (see Fig. 26) that our incremental stack-splitting system obtains higher speed-ups than using the delay termination version.

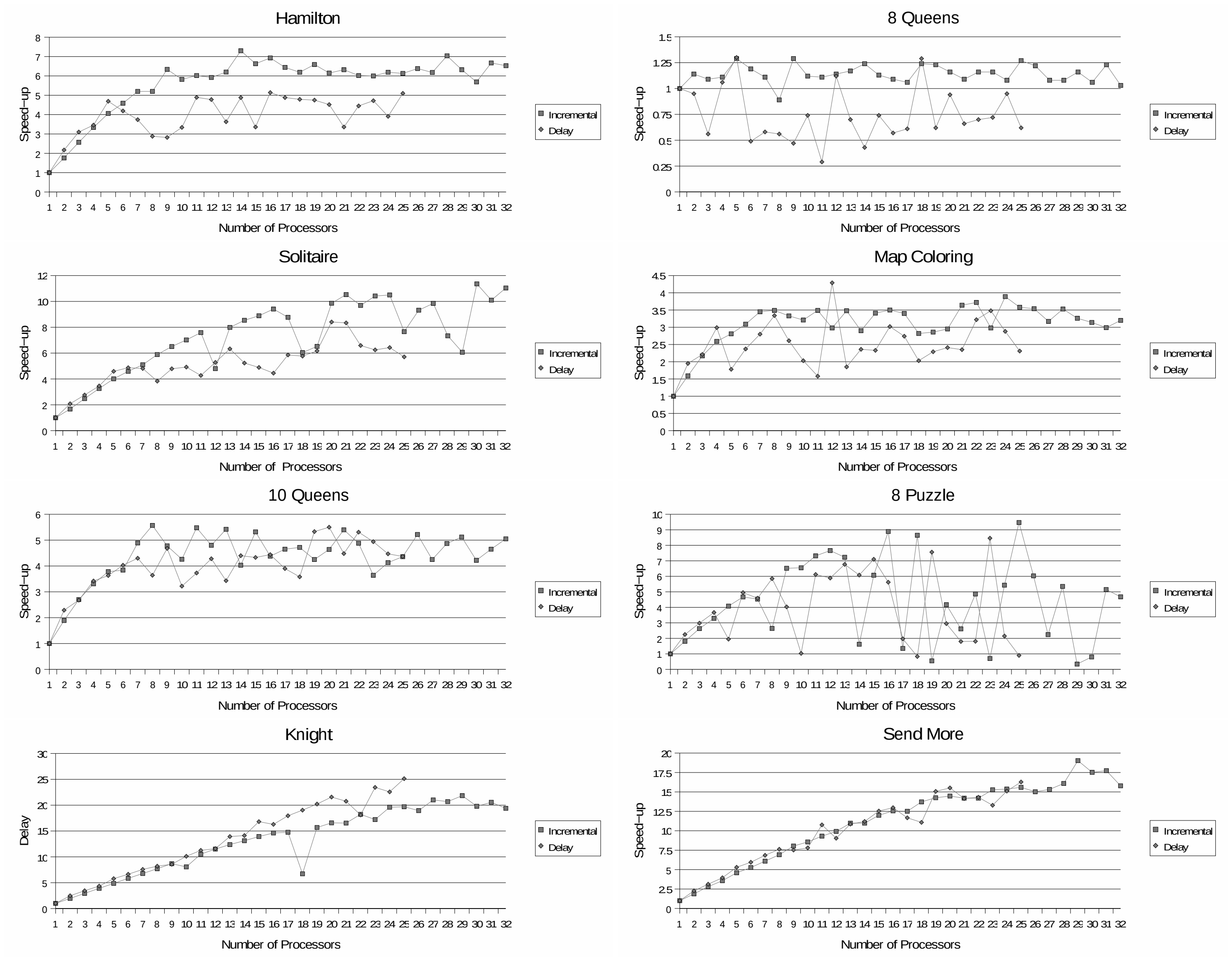


Figure 26: Incremental Stack-Splitting vs. Delay Termination

8.3 Order-sensitive Computations

We implemented the techniques to handle $\mathcal{OSC}$ described in Section 7 in our PALS Prolog system and tested it with the following benchmarks. *Stable* is an engine to compute stable models of a logic program. *Knight* finds a path of knight-moves on a chess-board visiting every square on the board. *9 Costas* computes the Costas sequences of length 9. *Hamilton* finds a closed path through a graph such that all the nodes of the graph are visited once. *10 Queens* places a number of queens on a chess-board so that no two queens attack each other. *Map Coloring* colors a planar map. These benchmarks compute all solutions and execute side-effect predicates, e.g., *write* to describe the computations.

Figs. 27 and 28 show the speedups obtained by this technique under the label *side-effect*. The figures also show the speedups obtained when running these benchmarks without treating the *write* predicate as a side-effect but using the modified stack splitting approach described above. Two main observations arise from these experiments. First of all, the speedups obtained using the modified scheduling scheme are not that different from those observed in our previous experiments [36]; this means that the novel splitting strategy does not deteriorate the parallel performance of the system. For benchmarks with substantial running time and with the fewest number of printed solutions (*Knight*, *Stable*) the speedups are very good and close to the speedups obtained without handling $\mathcal{OSC}$. We also observe that for benchmarks with smaller running time but larger number of side-effects (*Hamilton*) the speedups are still good but less close to the speedups obtained without side-effects. Note that for benchmarks with small running time and the greatest number of printed solutions (*Map Coloring*, *10 Queens*), the speedups deteriorate significantly and may be less than 1. This is not surprising; the presence of large numbers of side-effects (proportional to the number of solutions) implies the introduction of a large sequential component in the computation, leading to reduced speedups. *9 Costas* has the largest number of solutions, but its speedups are good.

Benchmark	**Timings**	**Number of solutions**
9 Costas	412.579	760
Knight	159.950	60
Stable	62.638	2
10 Queens	4.572	724
Hamilton	3.175	80
Map Coloring	1.113	2594

Table 5: Benchmarks (Time in sec.)

The results obtained are consistent with our belief that DMP implementations should be used for programs with coarse-grained parallelism and a modest number of $\mathcal{OSC}$. Coarse-grained computations are even more important if we want to handle large numbers of side-effects where it is necessary that the $\mathcal{OSC}$ be spaced far apart. For programs with small-running times there is not enough work to offset the cost of exploiting parallelism and even less for handling $\mathcal{OSC}$. Nevertheless, our system is reasonably efficient given that it produces good speedups for large and medium size benchmarks with even a considerable number of $\mathcal{OSC}$, and produces no slow downs except for benchmarks with huge numbers of side-effects and small running times. Even in presence of $\mathcal{OSC}$, the parallel overhead observed is substantially low—on average 6.5% and seldomly over 10%. Fig. 29 compares with the speedups for some benchmarks obtained using a variant of the MUSE system [1] on SMP. The results highlight the fact that, for benchmarks with significant running time, our methodology is capable of approximating

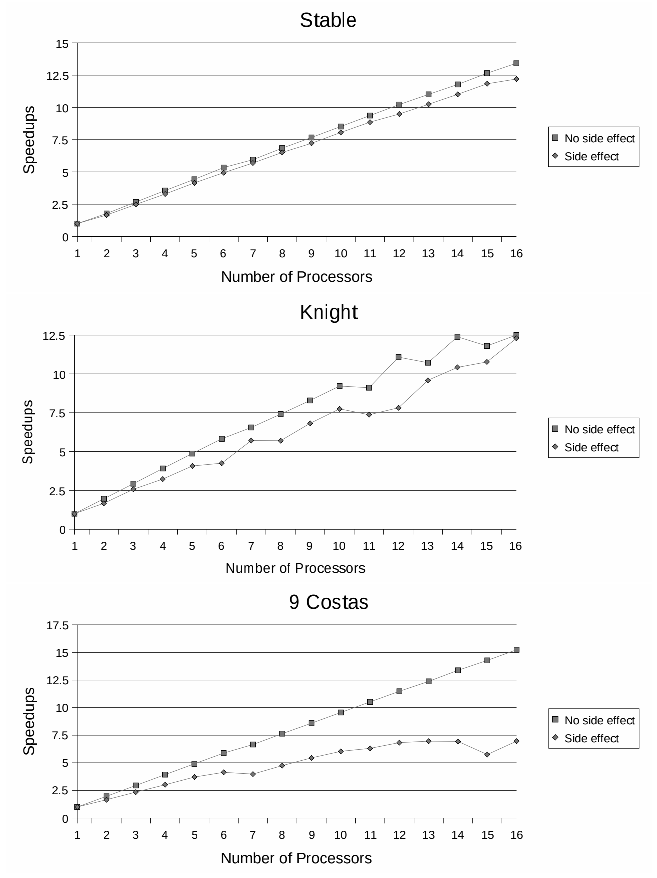


Figure 27: No Side-Effects vs. Side-Effects (1)

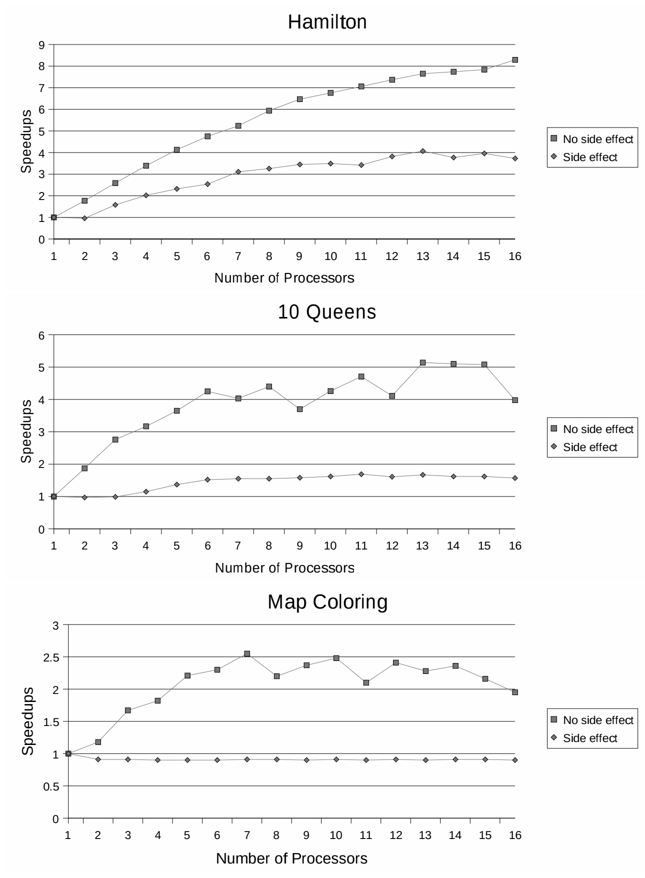


Figure 28: No Side-Effects vs. Side-Effects (2)

the best behavior on SMPs.

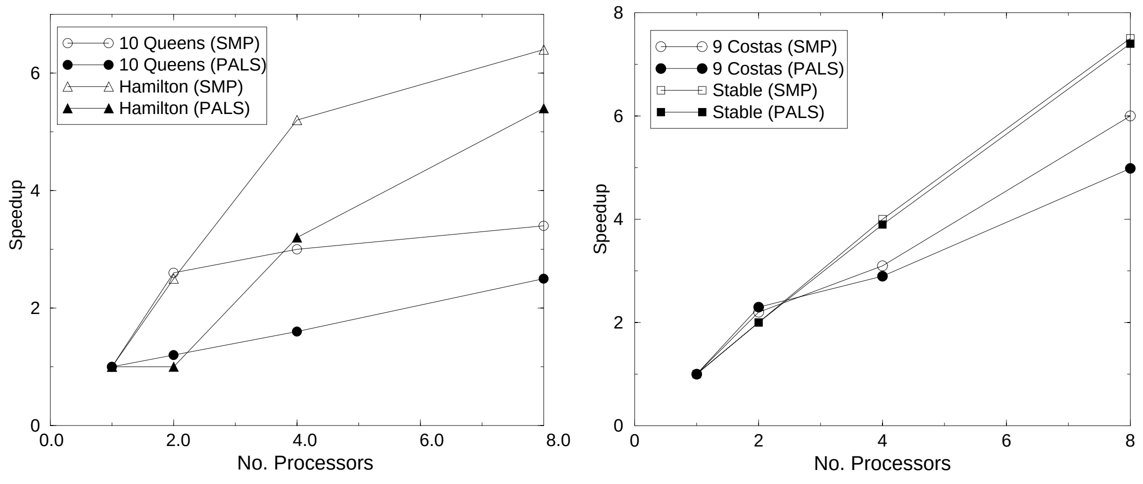


Figure 29: Comparison with MUSE

9 Conclusion and Related Work

In this paper, we presented a technique called stack-splitting for implementing OP and discussed its advantages and disadvantages. We show how stack-splitting can be extended to incremental stack-splitting which incrementally copies the difference of two stacks. Implementations on both a shared memory multiprocessor and a distributed memory multiprocessor were realized and reported. Our DMP implementation is the first ever implementation of a Prolog system on a Beowulf architecture.

Stack-splitting is an extension of stack-copying. Its main advantage, compared to other techniques for implementing OP, is that it allows large grain-sized work to be picked up by idle processors and executed efficiently without incurring excessive communication overhead. The technique bears some similarity to the Delphi model [9] used in parallel execution of Prolog (the Delphi model was not the inspiration for our stack-splitting technique), where computation leading to a goal with multiple alternatives is replicated in multiple processors, and each processor chooses a different alternative when that goal is reached. Instead of recomputing we use stack-copying, which, we believe, is more efficient. We also showed how stack-splitting can be used for implementing other search-based systems (e.g., non-monotonic reasoning systems under stable models semantics).

Distributed implementations of Prolog have been proposed by several researchers [12, 3, 8]. However, none of these systems are very effective in producing speedups over a wide range of benchmarks. Foong's system [12] and Castro et al's system [8] are based directly on stack-copying and generate communication overhead due to the shared choice-points (no real implementation exist for the two of them). Araujo's system uses recomputation [9] rather than stack-copying. Using recomputation for maintaining multiple environments is inherently inferior to stack-copying. The stack frames that are copied in the stack-copying technique capture the effect of a computation. In the recomputation technique these stack-frames are reproduced by re-running the computation. A computation may run for hours and yet produce only a single stack frame (e.g., a tail-recursive computation). Distributed implementations of Prolog have been developed on Transputer systems (The Opera System [7] and the system of Benjumea and Troya [6]). Of these, Benjumea's system has produced quite good results. However, both the Opera system and the Benjumea's system have been developed on now-obsolete Transputer hardware, and, additionally, both rely on a stack-copying mechanism which will produce poor performance in programs where the task-granularity is small. Finally, the idea of stack-splitting bears some similarities with some of the loop transformation techniques which are commonly adopted for parallelization of imperative programming languages, such as loop fission, loop tiling, and index set splitting [40].

Acknowledgments

Thanks to C. Geyer, L. Castro, V. Santos Costa, F. Silva, M. Carro, and M. Hermenegildo for discussions on implementation of distributed LP systems.

References

[1] K.A.M. Ali and R. Karlsson. The MUSE Approach to Or-Parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, 1990.

[2] L. Araujo. Full Prolog on a Distributed Architecture. In *Euro-Par*, pages 1173–1180. Springer Verlag, 1997.

[3] L. Araujo and J. Ruz. A Parallel Prolog System for Distributed Memory. *Journal of Logic Programming*, 33(1):49–79, 1998.

[4] H. Babu. Porting muse on ipsc860. Master's thesis, New Mexico State University, 1996.

[5] A.J. Beaumont and D. H. D. Warren. Scheduling Speculative Work in Or-Parallel Prolog Systems. In D. S. Warren, editor, *Proceedings of the International Conference on Logic Programming*, pages 135–149, Cambridge, MA, 1993. MIT Press.

[6] V. Benjumea and J.M. Troya. An OR Parallel Prolog Model for Distributed Memory Systems. In M. Bruynooghe and J. Penjam, editors, *International Symposium on Programming Languages Implementations and Logic Programming*, pages 291–301, Heidelberg, 1993. Springer Verlag.

[7] J. Briat, M. Favre, C. Geyer, and J. Chassin de Kergommeaux. OPERA: Or-Parallel Prolog System on Supernode. In P. Kacsuk and M. Wise, editors, *Implementations of Distributed Prolog*, pages 45–64. J. Wiley & Sons, New York, 1992.

[8] L.F. Castro, V. Santos Costa, C.F.R. Geyer, F. Silva, P.K. Vargas, and M.E. Correia. DAOS: Scalable And-Or Parallelism. In D. Pritchard and J. Reeve, editors, *Proceedins of EuroPar*, pages 899–908, Heidelberg, 1999. Springer Verlag.

[9] W.F. Clocksin and H. Alshawi. A Method for Efficiently Executing Horn Clause Programs Using Multiple Processors. *New Generation Computing*, 5:361–376, 1988.

[10] J.S. Conery. Binding Environments for Parallel Logic Programs in Nonshared Memory Multiprocessors. In *International Symposium on Logic Programming*, pages 457–467. San Francisco, Ieee Computer Society, August 1987.

[11] R. Finkel, V. Marek, N. Moore, and M. Truszczyński. Computing Stable Models in Parallel. In A. Provetti and S.C. Tran, editors, *Proceedings of the AAAI Spring Symposium on Answer Set Programming*, pages 72–75, Cambridge, MA, 2001. AAAI/MIT Press.

[12] W-K. Foong. *Combining and- and or-parallelism in Logic Programs: a distributed approach.* PhD thesis, University of Melbourne, 1995.

[13] S. Ganguly, A. Silberschatz, and S. Tsur. A Framework for the Parallel Processing of Datalog Queries. In H. Garcia-Molina and H. Jagadish, editors, *Proceedings of ACM SIGMOD Conference on Management of Data*, New York, 1990. ACM Press.

[14] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programs. In *International Symposium on Logic Programming*, pages 1070–1080. MIT Press, 1988.

[15] G. Gupta and B. Jayaraman. Analysis of Or-parallel Execution Models. *ACM Transactions on Programming Languages and Systems*, 15(4):659–680, 1993.

[16] G. Gupta and E. Pontelli. Last Alternative Optimization for Or-parallel Logic Programming Systems. In *Eight International Symposium on Parallel and Distributed Processing.* IEEE Computer Society, 1996.

[17] G. Gupta and E. Pontelli. Optimization Schemas for Parallel Implementation of Nondeterministic Languages and Systems. In *International Parallel Processing Symposium*, Los Alamitos, CA, 1997. IEEE Computer Society.

[18] G. Gupta and E. Pontelli. Stack-splitting: A Simple Technique for Implementing Or-parallelism and And-parallelism on Distributed Machines. In *International Conference on Logic Programming*, pages 290–304. MIT Press, 1999.

[19] G. Gupta, E. Pontelli, M. Carlsson, M. Hermenegildo, and K.M. Ali. Parallel Execution of Prolog Programs: a Survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, 2001.

[20] B. Hausman, A. Ciepielewski, and A. Calderwood. Cut and Side-Effects in Or-Parallel Prolog. In ICOT Staff, editor, *International Conference on Fifth Generation Computer Systems*, pages 831–840, Tokyo, Japan, November 1988. Springer Verlag.

[21] R.A. Kowalski. *Logic for Problem Solving.* Elsevier North-Holland Inc., Amsterdam, 1979.

[22] V. Kumar and L.N. Kanal. Parallel Depth-First Search on Muliprocessors. *International Journal of Parallel Programming*, 16(6):479–499, 1987.

[23] T-H. Lai and S. Sahni. Anomalies in Parallel Branch-and-Bound Algorithms. *Communications of the ACM*, 27(6):594–602, 1984.

[24] E. Lusk, R. Butler, T. Disz, R. Olson, R. Stevens, D. H. D. Warren, A. Calderwood, P. Szeredi, P. Brand, M. Carlsson, A. Ciepielewski, B. Hausman, and S. Haridi. The Aurora Or-parallel Prolog System. *New Generation Computing*, 7(2/3):243–271, 1990.

[25] J. Montelius and K. Ali. A Parallel Implementation of AKL. *New Generation Computing*, 14(1):31–52, 1996.

[26] L. Perron. Search Procedures and Parallelism in Constraint Programming. In J. Jaffar, editor, *Proceedings of the International Conference on Principles and Practice of Constraint Programming*, volume 1713 of *LNCS*, pages 346–360, Heidelberg, 1999. Springer Verlag.

[27] E. Pontelli and O. El-Kathib. Construction and Optimization of a Parallel Engine for Answer Set Programming. In I. V. Ramakrishnan, editor, *Practical Aspects of Declarative Languages*, volume 1990 of *LNCS*, pages 288–303, Heidelberg, 2001. Springer Verlag.

[28] D. Ranjan, E. Pontelli, and G. Gupta. On the Complexity of Or-Parallelism. *New Generation Computing*, 17(3):285–308, 1999.

[29] D. Ranjan, E. Pontelli, and G. Gupta. Data Structures for Order-Sensitive Predicates in Parallel Nondeterministic Systems. *ACTA Informatica*, 37(1):21–43, 2000.

[30] R. Rocha, F. Silva, and V. Santos Costa. YapOr: an Or-Parallel Prolog System based on Environment Copying. In *LNAI 1695, Proceedings of EPPIA'99: The 9th Portuguese Conference on Artificial Intelligence*, pages 178–192. Springer-Verlag LNAI Series, September 1999.

[31] C. Schulte. Compairing Trailing and Copying for Constraint Programming. In *International Conference on Logic Programming*, pages 275–289. MIT Press, 1999.

[32] C. Schulte. Parallel Search Made Simple. In N. Beldiceanu et al., editor, *Proceedings of Techniques for Implementing Constraint Programming Systems, Post-conference workshop of CP 2000*, number TRA9/00, pages 41–57, University of Singapore, 2000.

[33] F. Silva and P. Watson. Or-Parallel Prolog on a Distributed Memory Architecture. *Journal of Logic Programming*, 43(2):173–186, 2000.

[34] F. Silva et al. YapDss: an Or-Parallel Prolog System for Scalable Beowulf Clusters. Technical report, University of Porto, 2003.

[35] P. Szeredi. Using dynamic predicates in an or-parallel prolog system. In V. Saraswat and K. Ueda, editors, *Proceedings of the International Logic Programming Symposium*, pages 355–371, Cambridge, MA, October 1991. MIT Press.

[36] K. Villaverde. *An Efficient Methodology to Exploit Or-parallelism on Distributed Memory Systems.* PhD thesis, New Mexico State University, 2002.

[37] K. Villaverde, E. Pontelli, G. Gupta, and H. Guo. Incremental Stack Splitting Mechanisms for Efficient Parallel Implementation of Search-based Systems. In *International Conference on Parallel Processing.* IEEE Computer Society, 2001.

[38] K. Villaverde, E. Pontelli, G. Gupta, and H. Guo. PALS: An Or-parallel Implementation of Prolog on Bewoulf Architectures. In *Procs. International Conference on Logic Programming.* Springer Verlag, 2001.

[39] K. Villaverde, E. Pontelli, G. Gupta, and H. Guo. A Methodology for Order-sensitive Execution of Non-deterministic Languages on Beowulf Platforms. In *Euro-Par*, 2003.

[40] M. Wolfe. *High Performance Compiler for Parallel Computing.* Addison Wesley, 1996.

[41] O. Wolfson and A. Silberschatz. Distributed Processing of Logic Programs. In H. Boral and P. Larson, editors, *Proceedings of the SIGMOD International Conference on Management of Data*, pages 329–336, New York, 1988. ACM, ACM Press.

[42] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, 1991.